A Deductive Proof System for Multithreaded Java with Exceptions*

July 23, 2003

Erika Ábrahám[1], Frank S. de Boer[2],
Willem-Paul de Roever[1], and Martin Steffen[1]

[1] Christian-Albrechts-University Kiel, Germany
[2] CWI Amsterdam, The Netherlands

* Part of this work has been financially supported by IST project Omega (IST-2001-33522) and NWO/DFG project Mobi-J (RO 1122/9-1, RO 1122/9-2).

Table of Contents

1 Introduction

Since the *Java* language is increasingly used also in safety-critical applications, the development of verification techniques for *Java* programs becomes more and more important. *Java* has several interesting and challenging features likes object-orientation, inheritance, and exception handling. Furthermore, *Java* integrates concurrency via its `Thread`-class, allowing for a multithreaded flow of control.

To reason about *safety* properties of multithreaded *Java* programs, this work introduces a tool-supported *assertional proof method* for a concurrent sublanguage of *Java*. The language includes dynamic object creation, method invocation, object references with aliasing, *concurrency*, Java's *monitor discipline*, and *exception handling*, but excludes *inheritance* and *subtyping*. The concurrency model includes shared-variable concurrency via instance variables, coordination via reentrant synchronization monitors, synchronous message passing, and dynamic thread creation.

A program specifies a set of classes, where each class declares its own methods and instance variables. The behavior of a *Java* program results from the concurrent execution of methods.

To support a clean interface between internal and external object behavior, we exclude qualified references to instance variables. I.e., the values of instance variables of an object can be accessed and modified only within the object. As a consequence, shared-variable concurrency is caused by simultaneous execution within a single object, only, but not across object boundaries.

In order to capture program behavior in a modular way, the assertional logic and the proof system are formulated at two levels, a local and a global one.

The local assertion language describes the internal object behavior. The global behavior, including the communication topology of objects, is expressed in the global language. As in the Object Constraint Language (OCL) [WK99], properties of object-structures are described in terms of a navigation or dereferencing operator.

The assertional proof system is formulated in terms of *proof outlines* [OG76], i.e., of programs augmented by auxiliary variables and annotated with Hoare-style assertions [Flo67,Hoa69]. The satisfaction of the program properties specified by the assertions is guaranteed by the verification conditions of the proof system. The *initial correctness* conditions cover satisfaction of the properties in the initial program configuration. The execution of a single method body in isolation is captured by standard *local correctness* conditions, using the local assertion language. Interference between concurrent method executions is covered by the *interference freedom test* [OG76,LG81], formulated also in the local language. It has especially to accommodate for reentrant code and the specific synchronization mechanism. Possibly affecting more than one instance, communication and object creation is treated in the *cooperation test*, using the global language. The communication can take place within a single object or between different objects. As these cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules in [AFdR80] and in [LG81] for CSP.

Our proof method is *modular* in the sense that it allows for separate interference freedom and cooperation tests. This modularity, which in practice simplifies correctness proofs considerably, is obtained by disallowing the assignment of the result of communication and object creation to instance variables. Clearly, such assignments can be avoided by additional assignments to fresh local variables and thus at the expense of new interleaving points. This restriction could be released, without loosing the mentioned modularity, but it would increase the complexity of the proof system.

Computer-support is given by the tool Verger (*VERification condition GEneratoR*), taking a proof outline as input and generating the verification conditions as output. We use the interactive theorem prover PVS [ORS92] to verify the conditions, for which we only need to encode the semantics of the assertion language.

To transparently describe the proof system, we present it incrementally in four stages: We start with a proof method for a *sequential* sublanguage of *Java*, allowing for dynamic object creation and method invocation. This first stage shows how to handle activities of a single *thread* of execution. In the second stage we additionally allow dynamic thread creation, leading to *multithreaded* execution. The corresponding proof system extends the one for the sequential case with conditions handling dynamic thread creation and the new interleaving aspects. We integrate *Java*'s *monitor synchronization* mechanism in the third stage. Finally, we include *Java*'s *exception handling* in the last stage. We also show how to express *deadlock freedom*, and give some examples. The proof system is shown to be sound and complete.

This incremental development shows how the proof system can be extended stepwise to deal with additional features of the programming language. Further extensions by, for example, the concepts of inheritance and subtyping are topics for future work.

1.1 Related work

This work extends earlier results. In [ÁMdB00] we develop a proof system for a concurrent sublanguage of *Java*, but without reentrant monitors. Reentrant synchronization was incorporated in [ÁMdBdRS02b]; the work [ÁdBdRS03b] integrates also *Java*'s *monitor methods* wait, notify, and notifyAll. An incremental description of the proof system, starting with a sequential language and stepwise adding additional language features, but excluding exception handling, is given in [ÁMdBdRS02a]. In [ÁMdBdRS02a] we also introduce proof conditions for deadlock freedom. We discuss the proof system also in [ÁMdBdRS01] and in [ÁdBdRS03c]. We formalize the semantics of our programming language in a compositional manner in [ÁdBdRS03a]. This work extends the above ones by including exception handling.

The semantical foundations of *Java* have been thoroughly studied ever since the language gained widespread popularity (see e.g. [AF99,SSB01,CKRW99]). The research concerning *Java*'s proof theory mainly concentrated on various aspects of *sequential* sub-languages. To the best of our knowledge, our work defines the first sound and complete assertional proof method for a multithreaded sublanguage of *Java* including its monitor discipline and exception handling.

De Boer [dB99] presents a sound and complete proof system in weakest precondition formulation for a parallel object-based language, i.e., without inheritance and subtyping, and also without reentrant method calls. Later work [PdB03,dBP03,dBP02] includes more features, especially catering for Hoare logic for inheritance and subtyping.

The aim of the work in the LOOP project (Logic of Object-Oriented Programming) [Loo01] is to specify and verify properties of classes in object-oriented languages. The project research concentrates on a sequential subpart of *Java*; the main focus of application is *JavaCard*.

A compiler [vdBJ02] translates programs and their specifications into *PVS* [JvdBH$^+$98a,JvdBH$^+$98b] and *Isabelle/HOL* [vdBHJP00]. The translation is based on the embedding of a denotational semantics of the sequential *Java* subset into Higher Order Logic (HOL). Soundness of the representation is shown in [Hui01]. LOOP specifications formalized in *JML* are represented in HOL by a set of proof rules [JP01]. Jacobs presents also a coalgebraic view of exceptions in [Jac01]. Modeling inheritance in higher order logic is the topic of [HJ00]. The LOOP tool and methodology has been applied to several case studies; see e.g. [PvdBJ01,PvdBJ00,vdBJP01,HJvdB01,JKW03].

Instead of the denotational semantics, our work is based on an operational semantics. Though research within the LOOP project deals with many of the complexities of *Java*, they don't handle recursive calls and concurrency, and don't investigate completeness.

The project Bali [Bal03] is concerned with the formalization of various aspects of *Java* in the theorem prover *Isabelle/HOL* [Pau93]. Nipkow and von Oheimb [NvO98,vON99] prove type soundness of their $Java_{light}$ subset, a large sequential sublanguage of *Java*. They formalize its abstract syntax, type system, and well-formedness conditions. Instead of the denotational semantics in works of the LOOP project, they develop an operational semantics. Based on this formalization, they express and prove type soundness within the theorem prover *Isabelle/HOL*. To complement the operational semantics of $Java_{light}$, von Oheimb presents an axiomatic semantics [vO00a,vO00b], and proves soundness and completeness of the latter with respect to the operational semantics.

With μ*Java*, Nipkow et al. [NvOP00] offer an *Isabelle/HOL* embedding of *Java*'s imperative core with classes. They present a static and a dynamic semantics of the language both at the *Java* level and the *JVM* level.

Based on [NvOP00], von Oheimb [vO01] presents a Hoare-style calculus for a *JavaCard* subset and proves soundness and completeness in *Isabelle/HOL*. Nipkow [Nip02] selects some of the technically difficult language features and deals with their Hoare logic in isolation. The combination of [vO01] and [Nip02] in one language (NanoJava) is formulated in [vON02].

In contrast to our approach, the Bali project aims to cover only sequential subsets of *Java*. Furthermore, they use a semantic representation of assertions; program execution is specified by state transformations. Our proof system uses a syntactic representation, and substitution operators instead of state transformations.

Similarly to our proof system, also Poetzsch-Heffter and Müller use a syntactical representation of assertions [PH97a,PH97b,PHM98,PHM99]. They develop a Hoare-style programming logic for a sequential kernel of *Java*, featuring interfaces, subtyping, and inheritance. Translating the operational and the axiomatic semantics into the HOL theorem prover allows a computer-assisted soundness proof. Neither this group deals with concurrent sublanguages of *Java*.

1.2 Overview

The work is organized as follows: Section 2 describes syntax and semantics of a sequential sublanguage of *Java*. After introducing the assertional logic, we present a proof system for the sequential case. Section 3 extends the results to a concurrent sublanguage. The language introduced in Section 4 includes *Java*'s monitor synchronization mechanism. Section 5 covers also exception handling. The verification conditions in the above sections are formulated as standard Hoare-triples. Section 6 defines the formal semantics of Hoare-triples, given by means of a weakest precondition calculus, and reformulates the verification conditions. Soundness and completeness are discussed in Section 7. Section 8 shows how we can prove deadlock freedom, and gives some examples. Section 9 contains some concluding remarks. The appendix contains proofs of soundness and completeness.

2 The sequential language

In this section we introduce a sequential sublanguage $Java_{seq}$ of *Java*. We define its syntax in Section 2.1, and its semantics in Section 2.2. After defining the assertion language in Section 2.3, we introduce a proof system for verifying safety properties of the language in Section 2.4.

Programs, as in *Java*, are given by a collection of classes containing instance variable and method declarations. *Instances* of the classes, i.e., *objects,* are dynamically created, and communicate via *method invocation,* i.e., synchronous message passing.

We ignore in $Java_{seq}$ the issues of *concurrency*, *inheritance*, and consequently subtyping, overriding, and late-binding. For simplicity, we neither allow method *overloading*, i.e., we require that each method name is assigned a unique list of formal parameter types and a return type. In short, being concerned with the verification of the run-time behavior, we assume a simple *monomorphic* type discipline for $Java_{seq}$.

2.1 Syntax

$Java_{seq}$ is a strongly typed language; besides class types c, it supports booleans Bool and integers Int as primitive types, and pairs $t \times t$ and lists $\mathsf{list}\, t$ as composite types. The type of methods without return value is Void. Since $Java_{seq}$ is strongly typed, all program constructs of the abstract syntax are silently assumed to be well-typed. In other words, we work with a type-annotated abstract syntax where we omit the explicit mentioning of types when this causes no confusion.

For each type, the corresponding value domain is equipped with a standard set of operators with typical element f. Each operator f has a unique type $t_1 \times \cdots \times t_n \to t$ and a fixed interpretation f, where constants are operators of zero arity. Apart from the standard repertoire of arithmetical and boolean operations, the set of operators also contains operations on tuples and sequences like projection, concatenation, etc.

For variables, we notationally distinguish between *instance variables* $x \in IVar$ and *local (temporary) variables* $u \in TVar$. Instance variables hold the state of an object and exist throughout the object's lifetime. Local variables are stack-allocated; they play the role of formal parameters and variables of method definitions and only exist during the execution of the method to which they belong. We use $Var = IVar \,\dot{\cup}\, TVar$ for the set of program variables with typical element y, where $\dot{\cup}$ is the disjoint union operator.

The abstract syntax is summarized in Table 1. It slightly differs from *Java* syntax. Though we use the abstract syntax for the theoretical part of this work, our tool supports *Java* syntax.

$$
\begin{array}{rcl}
exp & ::= & x \mid u \mid \mathsf{this} \mid \mathsf{null} \mid \mathsf{f}(exp,\ldots,exp) \\
exp_{ret} & ::= & \epsilon \mid exp \\
stm & ::= & x := exp \mid u := exp \mid u := \mathsf{new}^c \\
 & \mid & u := exp.m(exp,\ldots,exp) \mid exp.m(exp,\ldots,exp) \\
 & \mid & \epsilon \mid stm; stm \mid \mathsf{if}\ exp\ \mathsf{then}\ stm\ \mathsf{else}\ stm\ \mathsf{fi} \mid \mathsf{while}\ exp\ \mathsf{do}\ stm\ \mathsf{od}\ldots \\
meth & ::= & m(u,\ldots,u)\{\ stm; \mathsf{return}\ exp_{ret}\} \\
meth_{\mathsf{run}} & ::= & \mathsf{run}()\{\ stm; \mathsf{return}\ \} \\
class & ::= & \mathsf{class}\ c\{meth\ldots meth\} \\
class_{\mathsf{main}} & ::= & c\{meth\ldots meth\ meth_{\mathsf{run}}\} \\
prog & ::= & \langle class\ldots class\ class_{\mathsf{main}}\rangle
\end{array}
$$

Table 1. $Java_{seq}$ abstract syntax

Besides using instance and local variables, *expressions* $exp \in Exp$ are built from the self-reference this, the empty reference null, and from subexpressions using the given operators. We use e as typical element for expressions. To support a clean interface between internal and external object behavior, $Java_{seq}$ does not allow qualified references to instance variables. Note that all expressions of the language are side-effect free, i.e., their evaluation does not modify the program state. Only the execution of statements may have such an effect.

As *statements* $stm \in Stm$, we allow assignments, object creation, method invocation, and standard control constructs like sequential composition, conditional statements, and iteration. We write ϵ for the empty statement.

A *method* definition $m(u_1,\ldots,u_n)\{stm; \mathsf{return}\ e_{ret}\}$ specifies the method's name m, a list of formal parameters $u_1,\ldots,u_n$, and a method body of the form $stm; \mathsf{return}\ e_{ret}$, i.e., we require that method bodies are terminated by a single return statement, giving back the control and possibly a return value. The set $Meth_c$ contains the methods of class c. We denote the body of method m of class c by $body_{m,c}$. Sometimes we explicitly mention the types of formal parameters and of the return value in *Java*-style $t\ m(t_1\ u_1,\ldots,t_n\ u_n)\{body_{m,c}\}$.

A *class* is defined by its name c and its methods, whose names are assumed to be distinct. A *program,* finally, is a collection of class definitions having different class names, where $class_{\mathsf{main}}$ defines by its run-method the entry point of the program execution. We call the body of the run-method of the main class the *main statement* of the program.[3] The run-method cannot be called.

The set $IVar_c$ of instance variables of a class c is given implicitly by the instance variables occurring in the class; the set of local variables of method declarations is given similarly. In the examples we explicitly define variables in *Java*-style.

[3] In *Java*, the entry point of a program is given by the static main-method of the main class. Relating the abstract syntax to that of *Java*, we assume that the main class is a `Thread`-class whose main-method just creates an instance of the main class and starts its thread. The reason to make this restriction is, that *Java*'s main-method is static, but our proof system does not support static methods and variables.

Besides the mentioned simplifications on the type system, we impose for technical reasons the following restrictions: We require that method invocation statements contain only local variables, i.e., that none of the expressions $e_0, \ldots, e_n$ in a method invocation $e_0.m(e_1, \ldots, e_n)$ contains instance variables. Furthermore, formal parameters must not occur on the left-hand side of assignments. These restrictions imply that during the execution of a method the values of the actual and formal parameters are not changed. Finally, the result of object creation and method invocation may not be stored in instance variables. This restriction allows for a proof system with separated verification conditions for interference freedom and cooperation. It should be clear that it is possible to transform a program to adhere to this restrictions at the expense of additional local variables and thus new interleaving points. The above restrictions could be released, without loosing the mentioned modularity, but it would increase the complexity of the proof system.

2.2 Semantics

In this section, we define the *operational semantics* of Java_{seq}. After introducing the semantic domains, we describe states and configurations. The operational semantics is presented by transitions between program configurations.

States and configurations Let Val^t be the disjoint domains of the various types t. For class names c, the disjunct sets Val^c with typical elements $\alpha, \beta, \ldots$ denote infinite sets of *object identifiers*. The value of null of type c is $\mathit{null}^c \notin \mathit{Val}^c$. In general we will just write null, when c is clear from the context. We define Val^c_{null} as $\mathit{Val}^c \,\dot\cup\, \{\mathit{null}^c\}$, and correspondingly for compound types. The set of all possible non-null values $\bigcup_t \mathit{Val}^t$ is written as Val, and Val_{null} denotes $\bigcup_t \mathit{Val}^t_{null}$. Let $\mathit{Init} : \mathit{Var} \to \mathit{Val}_{null}$ be a function assigning an initial value to each variable $y \in \mathit{Var}$, i.e., null, false, and 0 for class, boolean, and integer types, respectively, and analogously for compound types, where sequences are initially empty. We define $\mathsf{this} \notin \mathit{Var}$, such that the self-reference is not in the domain of Init.[4]

The configuration of a program consists of the set of existing objects and the values of their instance variables, and the configuration of the executing thread. Before formalizing the global configurations of a program, we define local states and local configurations. In the sequel we identify the occurrence of a statement in a program with the statement itself.

A *local state* $\tau \in \Sigma_{loc}$ of a method execution holds the values of the method's local variables and is modeled as a partial function of type $\mathit{TVar} \rightharpoonup \mathit{Val}_{null}$. We refer to local states of method m of class c by $\tau^{m,c}$. The initial local state $\tau^{m,c}_{init}$ assigns to each local variable u from its domain the value $\mathit{Init}(u)$. A *local configuration* $(\alpha, \tau, \mathit{stm})$ of a method of an object $\alpha \neq \mathit{null}$ specifies, in addition

[4] In *Java*, this is a "final" instance variable, which for instance implies, it cannot be assigned to.

to its local state τ, its point of execution represented by the statement stm. A *thread configuration* $\xi = (\alpha_0, \tau_0, stm_0)(\alpha_1, \tau_1, stm_1) \ldots (\alpha_n, \tau_n, stm_n)$ is a stack of local configurations, representing the chain of method invocations of the given thread. We write $\xi \circ (\alpha, \tau, stm)$ for pushing a new local configuration onto the stack.

Objects are characterized by their *instance states* $\sigma_{inst} \in \Sigma_{inst}$ of type $IVar\ \dot{\cup}\ \{\mathsf{this}\} \rightharpoonup Val_{null}$; we require that this is in the domain $dom(\sigma_{inst})$ of σ_{inst}. We write σ^c_{inst} to denote states of instances of class c. The semantics will maintain $\sigma^c_{inst}(\mathsf{this}) \in Val^c$ as invariant. The initial instance state $\sigma^{c,init}_{inst}$ assigns a value from Val^c to this, and to each of its remaining instance variables x the value $Init(x)$.

A *global state* $\sigma \in \Sigma$ of type $(\bigcup_c Val^c) \rightharpoonup \Sigma_{inst}$ stores for each currently *existing* object, i.e., an object belonging to the domain of σ, its instance state. The set of existing objects of type c in a state σ is given by $Val^c(\sigma)$, and $Val^c_{null}(\sigma) = Val^c(\sigma)\ \dot{\cup}\ \{null^c\}$. For the remaining types, $Val^t(\sigma)$ and $Val^t_{null}(\sigma)$ are defined correspondingly. We refer to the set $\bigcup_t Val^t(\sigma)$ by $Val(\sigma)$; $Val_{null}(\sigma)$ denotes $\bigcup_t Val^t_{null}(\sigma)$. The instance state of an object $\alpha \in Val(\sigma)$ is given by $\sigma(\alpha)$ with the invariant property $\sigma(\alpha)(\mathsf{this}) = \alpha$. We require that, given a global state, no instance variable in any of the existing objects refers to a non-existing object, i.e., $\sigma(\alpha)(x) \in Val_{null}(\sigma)$ for all classes c, objects $\alpha \in Val^c(\sigma)$, and instance variables $x \in IVar_c$. This will be an invariant of the operational semantics of the next section.

A *global configuration* $\langle T, \sigma \rangle$ describes the currently existing objects by the global state σ, where the set T contains the configuration of the executing thread. For the concurrent languages of the later sections, T will be the set of configurations of all currently executing threads. Analogously to the restriction on global states, we require that local configurations (α, τ, stm) in $\langle T, \sigma \rangle$ refer only to existing object identities, i.e., $\alpha \in Val(\sigma)$ and $\tau(u) \in Val_{null}(\sigma)$ for all variables u from the domain of τ; again this will be an invariant of the operational semantics. In the following, we write $(\alpha, \tau, stm) \in T$ if there exists a local configuration (α, τ, stm) within one of the execution stacks of T.

The semantic function $[\![\_]\!]^{-,-}_{\mathcal{E}} : (\Sigma_{inst} \times \Sigma_{loc}) \to (Exp \rightharpoonup Val_{null})$ evaluates in the context of an *instance local* state (σ_{inst}, τ) expressions containing variables from $dom(\sigma_{inst}) \cup dom(\tau)$: Instance variables x and local variables u are evaluated to $\sigma_{inst}(x)$ and $\tau(u)$, respectively; this evaluates to $\sigma_{inst}(\mathsf{this})$, and null has the *null*-reference as value, where compound expressions are evaluated by homomorphic lifting (see Table 2).

We denote by $\tau[u \mapsto v]$ the local state which assigns the value v to u and agrees with τ on the values of all other variables; $\sigma_{inst}[x \mapsto v]$ is defined analogously, where $\sigma[\alpha.x \mapsto v]$ results from σ by assigning v to the instance variable x of object α. We use these operators analogously for vectors of variables. We use $\tau[\vec{y} \mapsto \vec{v}]$ also for arbitrary variable sequences, where instance variables are untouched; $\sigma_{inst}[\vec{y} \mapsto \vec{v}]$ and $\sigma[\alpha.\vec{y} \mapsto \vec{v}]$ are analogous. Finally for global states, $\sigma[\alpha \mapsto \sigma_{inst}]$ equals σ except on α; note that in case $\alpha \notin Val(\sigma)$, the operation extends the set of existing objects by α, which has its instance state initialized to σ_{inst}.

$$\begin{aligned}
[\![x]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} &= \sigma_{inst}(x)\\
[\![u]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} &= \tau(u)\\
[\![\mathsf{this}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} &= \sigma_{inst}(\mathsf{this})\\
[\![\mathsf{null}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} &= null\\
[\![\mathsf{f}(e_1,\ldots,e_n)]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} &= f([\![e_1]\!]_{\mathcal{E}}^{\sigma_{inst},\tau},\ldots,[\![e_n]\!]_{\mathcal{E}}^{\sigma_{inst},\tau})
\end{aligned}$$

Table 2. Semantics of program expressions

Operational semantics The operational semantics of $Java_{seq}$ is given inductively by the rules of Table 3 as transitions between global configurations. The rules are formulated such a way that we can re-use them also for the concurrent languages of the later sections. Note that for the sequential language, the sets T in the rules are empty, since there is only one single thread in global configurations. We elide the rules for the remaining sequential constructs —sequential composition, conditional statement, and iteration— as they are standard.

Before having a closer look at the semantical rules for the transition relation $\longrightarrow$, let us start by defining the starting point of a program. The initial configuration $\langle T_0, \sigma_0\rangle$ of a program satisfies $dom(\sigma_0) = \{\alpha\}$, $\sigma_0(\alpha) = \sigma_{inst}^{c,init}[\mathsf{this} \mapsto \alpha]$, and $T_0 = \{(\alpha, \tau_{init}^{\mathsf{run},c}, body_{\mathsf{run},c})\}$, where c is the main class, and $\alpha \in Val^c$.

We call a configuration $\langle T, \sigma\rangle$ of a program *reachable* iff there exists a computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T, \sigma\rangle$ such that $\langle T_0, \sigma_0\rangle$ is the initial configuration of the program and $\longrightarrow^*$ the reflexive transitive closure of $\longrightarrow$. A local configuration $(\alpha, \tau, stm) \in T$ is *enabled* in $\langle T, \sigma\rangle$, if it can be executed, i.e., if there is a computation step $\langle T, \sigma\rangle \rightarrow \langle T', \sigma'\rangle$ executing stm in the local state τ and object α.

Assignments to instance or local variables update the corresponding state component, i.e., either the instance state or the local state (rules $\text{A}\textsc{ss}_{inst}$ and $\text{A}\textsc{ss}_{loc}$). Object creation by $u := \mathsf{new}^c$, as shown in rule $\text{N}\textsc{ew}$, creates a new object of type c with a fresh identity stored in the local variable u, and initializes the instance variables of the new object. Invoking a method extends the call chain by a new local configuration (rule $\text{C}\textsc{all}$). After initializing the local state and passing the parameters, the thread begins to execute the method body. When returning from a method call (rule $\text{R}\textsc{eturn}$), the callee evaluates its return expression and passes it to the caller which subsequently updates its local state. The method body terminates its execution and the caller can continue. We have similar rules not shown in the table for the invocation of methods without return value. The executing thread ends its lifespan by returning from the run-method of the initial object (rule $\text{R}\textsc{eturn}_{run}$).

2.3 The assertion language

In this section we introduce *assertions* to specify program properties. The assertion logic consists of a *local* and a *global* sublanguage. *Local* assertions describe

$$\frac{}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, x{:=}e; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau, stm)\}, \sigma[\alpha.x \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}]\rangle} \ \textsc{Ass}_{inst}$$

$$\frac{}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, u{:=}e; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau[u \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}], stm)\}, \sigma\rangle} \ \textsc{Ass}_{loc}$$

$$\frac{\beta \in Val^c \setminus Val(\sigma) \qquad \sigma_{inst} = \sigma_{inst}^{c,init}[\mathsf{this} \mapsto \beta] \qquad \sigma' = \sigma[\beta \mapsto \sigma_{inst}]}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, u{:=}\mathsf{new}^c; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau[u \mapsto \beta], stm)\}, \sigma'\rangle} \ \textsc{New}$$

$$\frac{\begin{array}{c} m(\vec{u})\{\ body\ \} \in Meth_c \\ \beta = [\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in Val^c(\sigma) \qquad \tau' = \tau_{init}^{m,c}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \end{array}}{\begin{array}{l} \langle T \dot\cup \{\xi \circ (\alpha, \tau, u := e_0.m(\vec{e}); stm)\}, \sigma\rangle \longrightarrow \\ \langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{receive}\ u; stm) \circ (\beta, \tau', body)\}, \sigma\rangle \end{array}} \ \textsc{Call}$$

$$\frac{\tau'' = \tau[u_{ret} \mapsto [\![e_{ret}]\!]_{\mathcal{E}}^{\sigma(\beta),\tau'}]}{\begin{array}{l} \langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{receive}\ u_{ret}; stm) \circ (\beta, \tau', \mathsf{return}\ e_{ret})\}, \sigma\rangle \longrightarrow \\ \langle T \dot\cup \{\xi \circ (\alpha, \tau'', stm)\}, \sigma\rangle \end{array}} \ \textsc{Return}$$

$$\frac{}{\langle T \dot\cup \{(\alpha, \tau, \mathsf{return})\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{(\alpha, \tau, \epsilon)\}, \sigma\rangle} \ \textsc{Return}_{run}$$

Table 3. $Java_{seq}$ operational semantics

instance local states, and are used to annotate methods in terms of their local variables and of the instance variables of the class to which they belong. *Global* assertions describe the global state, i.e., a whole system of objects and their communication structure.

To be able to argue about communication histories, represented as lists of objects, we add the type Object as the supertype of all classes into the assertion language. Note that we allow this type solely in the assertion language, but not in the programming language, thus preserving the assumption of monomorphism.

Syntax In the language of assertions, we introduce a countably infinite set $LVar$ of well-typed *logical variables* with typical element z, where we assume that instance variables, local variables, and this are not in $LVar$. We use $LVar^t$ for the set of logical variables of type t. Logical variables are used for quantification in both the local and the global language. Besides that, they are used as free variables to represent local variables in the global assertion language: To express a local property on the global level, each local variable in a given local assertion will be replaced by a fresh logical variable.

Table 4 defines the syntax of the assertion language. For readability, we use the standard syntax of first order logic in the theoretical part; the *Verger* tool supports an adaptation of *JML*.

Local expressions $exp_l \in LExp$ are expressions of the programming language possibly containing logical variables. The set of local expressions of type t is denoted by $LExp^t$. In abuse of notation, we use e, $e' \ldots$ not only for program expressions of Table 1, but also for typical elements of local expressions. *Local assertions* $ass_l \in LAss$, with typical elements $p, p', q, \ldots$, are standard logical formulas over boolean local expressions. We allow three forms of quantification over logical variables: Unrestricted quantification $\exists z.\ p$ is solely allowed for domains without object references, i.e., z is required to be of type Int, Bool, or compound types built from them. For reference types c, this form of quantification is not allowed, as for those types the existence of a value dynamically depends on the *global* state, something one cannot speak about on the local level, or more formally: Disallowing unrestricted quantification for object types ensures that the value of a local assertion indeed only depends on the values of the instance and local variables, but not on the global state. Nevertheless, one can assert the existence of objects on the local level satisfying a predicate, provided one is explicit about the set of objects to range over. Thus, the restricted quantifications $\exists z \in e.\ p$ and $\exists z \sqsubseteq e.\ p$ assert the existence of an element, respectively, the existence of a subsequence of a given sequence e, for which a property p holds.

Global expressions $exp_g \in GExp$, with typical elements $E, E', \ldots$, are constructed from logical variables, null, operator expressions, and qualified references $E.x$ to instance variables x of objects E. We write $GExp^t$ for the set of global expressions of type t. *Global assertions* $ass_g \in GAss$, with typical elements $P, Q \ldots$, are logical formulas over boolean global expressions. Unlike the local language, the meaning of the global one is defined in the context of a global state. Thus unrestricted quantification is allowed for all types and is interpreted to range over the set of *existing* values, i.e., the set of values $Val_{null}(\sigma)$ in a global configuration $\langle T, \sigma \rangle$.

$$
\begin{array}{rcll}
exp_l & ::= & z \mid x \mid u \mid \mathsf{this} \mid \mathsf{null} \mid \mathsf{f}(exp_l, \ldots, exp_l) & e \in LExp \\
ass_l & ::= & exp_l \mid \neg ass_l \mid ass_l \wedge ass_l & \\
 & \mid & \exists z.\ ass_l \mid \exists z \in exp_l.\ ass_l \mid \exists z \sqsubseteq exp_l.\ ass_l & p \in LAss \\[1ex]
exp_g & ::= & z \mid \mathsf{null} \mid \mathsf{f}(exp_g, \ldots, exp_g) \mid exp_g.x & E \in GExp \\
ass_g & ::= & exp_g \mid \neg ass_g \mid ass_g \wedge ass_g \mid \exists z.\ ass_g & P \in GAss
\end{array}
$$

Table 4. Syntax of assertions

We sometimes write quantification over t-typed values in the form $\forall (z : t).\ p$ to make the domain of the quantification explicit; we use the same notation also in the global language.

Semantics Next, we define the interpretation of the assertion language. The semantics is fairly standard, except that we have to cater for dynamic object creation when interpreting quantification.

Logical variables are interpreted relative to a logical environment $\omega \in \Omega$, a partial function of type $LVar \rightharpoonup Val_{null}$, assigning values to logical variables. We denote by $\omega[\vec{z} \mapsto \vec{v}]$ the logical environment that assigns the values $\vec{v}$ to the variables $\vec{z}$, and agrees with ω on all other variables. Similarly to local and instance state updates, the occurrence of instance and local variables in $\vec{z}$ is without effect. For a logical environment ω and a global state σ we say that ω refers only to values existing in σ, if $\omega(z) \in Val_{null}(\sigma)$ for all $z \in dom(\omega)$. This property matches with the definition of quantification which ranges only over existing values and *null*, and with the fact that in reachable configurations local variables may refer only to existing values or to *null*.

The semantic function $[\![\_]\!]_{\mathcal{L}}^{-,-,-}$ of type $(\Omega \times \Sigma_{inst} \times \Sigma_{loc}) \to (LExp \cup LAss \rightharpoonup Val_{null})$ evaluates local expressions and assertions in the context of a logical environment ω and an instance local state (σ_{inst}, τ) (cf. Table 5). The evaluation function is defined for expressions and assertions that contain only variables from $dom(\omega) \cup dom(\sigma_{inst}) \cup dom(\tau)$. The instance local state provides the context for giving meaning to programming language expressions as defined by the semantic function $[\![\_]\!]_{\mathcal{E}}$; the logical environment evaluates logical variables. An unrestricted quantification $\exists z.\, p$ with $z \in LVar^t$ is evaluated to true in the logical environment ω and instance local state (σ_{inst}, τ) if and only if there exists a value $v \in Val^t$ such that p holds in the logical environment $\omega[z \mapsto v]$ and instance local state (σ_{inst}, τ), where for the type t of z only Int, Bool, or compound types built from them are allowed. The evaluation of a restricted quantification $\exists z \in e.\, p$ with $z \in LVar^t$ and $e \in LExp^{\mathsf{list}\,t}$ is defined analogously, where the existence of an element in the sequence is required. An assertion $\exists z \sqsubseteq e.\, p$ with $z \in LVar^{\mathsf{list}\,t}$ and $e \in LExp^{\mathsf{list}\,t}$ states the existence of a subsequence of e for which p holds. In the following we also write $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ for $[\![p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true$. By $\models_{\mathcal{L}} p$ we express that $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ holds for arbitrary logical environments, instance states, and local states.

Since *global* assertions do not contain local variables and non-qualified references to instance variables, the global assertional semantics does not refer to instance local states but to global states. The semantic function $[\![\_]\!]_{\mathcal{G}}^{-,-}$ of type $(\Omega \times \Sigma) \rightharpoonup (GExp \cup GAss \rightharpoonup Val_{null})$, shown in Table 6, gives meaning to global expressions and assertions in the context of a global state σ and a logical environment ω. To be well-defined, ω is required to refer only to values existing in σ, and the expression respectively assertion may only contain free variables[5] from the domain of ω. Logical variables, null, and operator expressions are evaluated analogously to local assertions. The value of a global expression $E.x$ is given by the value of the instance variable x of the object referred to by the expression E. The evaluation of an expression $E.x$ is defined only if E refers to an object existing in σ. Note that when E and E' refer to the same object, that is, E and E' are *aliases*, then $E.x$ and $E'.x$ denote the same variable. The

[5] In global expressions $E.x$ we treat x as a bound variable.

$$
\begin{array}{rcl}
[\![z]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \omega(z) \\
[\![x]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \sigma_{inst}(x) \\
[\![u]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \tau(u) \\
[\![\mathsf{this}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \sigma_{inst}(\mathsf{this}) \\
[\![\mathsf{null}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & null \\
[\![\mathsf{f}(e_1,\ldots,e_n)]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & f([\![e_1]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau},\ldots,[\![e_n]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}) \\
([\![\neg p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}{=}true) & \text{iff} & ([\![p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}{=}false) \\
([\![p_1\wedge p_2]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}{=}true) & \text{iff} & ([\![p_1]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}{=}true \text{ and } [\![p_2]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}{=}true) \\
([\![\exists z.\, p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}{=}true) & \text{iff} & ([\![p]\!]_{\mathcal{L}}^{\omega[z\mapsto v],\sigma_{inst},\tau}{=}true \text{ for some } v\in Val_{null}) \\
([\![\exists z{\in}e.\, p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}{=}true) & \text{iff} & ([\![z{\in}e\wedge p]\!]_{\mathcal{L}}^{\omega[z\mapsto v],\sigma_{inst},\tau}{=}true \text{ for some } v{\in}Val_{null}) \\
([\![\exists z{\sqsubseteq}e.\, p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}{=}true) & \text{iff} & ([\![z{\sqsubseteq}e\wedge p]\!]_{\mathcal{L}}^{\omega[z\mapsto v],\sigma_{inst},\tau}{=}true \text{ for some } v{\in}Val_{null})
\end{array}
$$

Table 5. Local evaluation

semantics of negation and conjunction is standard. A quantification $\exists z.\ P$ with $z \in LVar^t$ evaluates to true in the context of ω and σ if and only if P evaluates to true in the context of $\omega[z\mapsto v]$ and σ, for some value $v \in Val^t_{null}(\sigma)$. Note that quantification over objects ranges over the set of *existing* objects and *null*, only.

$$
\begin{array}{rcl}
[\![z]\!]_{\mathcal{G}}^{\omega,\sigma} & = & \omega(z) \\
[\![\mathsf{null}]\!]_{\mathcal{G}}^{\omega,\sigma} & = & null \\
[\![\mathsf{f}(E_1,\ldots,E_n)]\!]_{\mathcal{G}}^{\omega,\sigma} & = & f([\![E_1]\!]_{\mathcal{G}}^{\omega,\sigma},\ldots,[\![E_n]\!]_{\mathcal{G}}^{\omega,\sigma}) \\
[\![E.x]\!]_{\mathcal{G}}^{\omega,\sigma} & = & \sigma([\![E]\!]_{\mathcal{G}}^{\omega,\sigma})(x) \\
([\![\neg P]\!]_{\mathcal{G}}^{\omega,\sigma} = true) & \text{iff} & ([\![P]\!]_{\mathcal{G}}^{\omega,\sigma} = false) \\
([\![P_1\wedge P_2]\!]_{\mathcal{G}}^{\omega,\sigma} = true) & \text{iff} & ([\![P_1]\!]_{\mathcal{G}}^{\omega,\sigma} = true \text{ and } [\![P_2]\!]_{\mathcal{G}}^{\omega,\sigma} = true) \\
([\![\exists z.\ P]\!]_{\mathcal{G}}^{\omega,\sigma} = true) & \text{iff} & ([\![P]\!]_{\mathcal{G}}^{\omega[z\mapsto v],\sigma} = true \text{ for some } v\in Val_{null}(\sigma))
\end{array}
$$

Table 6. Global evaluation

For a global state σ and a logical environment ω referring only to values existing in σ we write $\omega,\sigma \models_{\mathcal{G}} P$ when P is true in the context of ω and σ. We write $\models_{\mathcal{G}} P$ if P holds for arbitrary global states σ and logical environments ω referring only to values existing in σ.

To express a local property p in the global assertion language, we define the substitution $p[z/\mathsf{this}]$ by simultaneously replacing in p all occurrences of the self-reference this by the logical variable z, which is assumed not to occur in p, and transforming all occurrences of instance variables x into qualified references $z.x$. For notational convenience we view the local variables occurring in the global assertion $p[z/\mathsf{this}]$ as logical variables. Formally, these local variables are replaced by fresh logical variables. We write $P(z)$ for $p[z/\mathsf{this}]$, and similarly for expressions. For unrestricted quantifications $(\exists z'.\ p)[z/\mathsf{this}]$ the substitution applies to the assertion p. Local restricted quantifications are transformed into global unrestricted ones where the relations $\in$ and $\sqsubseteq$ are expressed at the global level as operators. The main cases of the substitution are defined as follows:

$$\begin{aligned}
\mathsf{this}[z/\mathsf{this}] &= z \\
x[z/\mathsf{this}] &= z.x \\
u[z/\mathsf{this}] &= u \\
(\exists z'.\ p)[z/\mathsf{this}] &= \exists z'.\ p[z/\mathsf{this}] \\
(\exists z' \in e.\ p)[z/\mathsf{this}] &= \exists z'.\ (z' \in e[z/\mathsf{this}] \wedge p[z/\mathsf{this}]) \\
(\exists z' \sqsubseteq e.\ p)[z/\mathsf{this}] &= \exists z'.\ (z' \sqsubseteq e[z/\mathsf{this}] \wedge p[z/\mathsf{this}])\ ,
\end{aligned}$$

where z is fresh.

This substitution will be used to combine properties of instance local states on the global level. The substitution preserves the meaning of local assertions, provided the meaning of the local variables is matchingly represented by the logical environment:

Lemma 1 (Lifting substitution). *Let σ be a global state, ω and τ a logical environment and local state, both referring only to values existing in σ. Let furthermore p be a local assertion containing local variables $\vec{u}$. If $\tau(\vec{u}) = \omega(\vec{u})$ and z a fresh logical variable, then*

$$\omega, \sigma \models_{\mathcal{G}} p[z/\mathsf{this}] \quad \textit{iff} \quad \omega, \sigma(\omega(z)), \tau \models_{\mathcal{L}} p\ .$$

The proof can be found in Appendix A.

2.4 The proof system

The proof system has to accommodate for dynamic object creation, aliasing, method invocation, and recursion. The following section defines how to augment and annotate programs resulting in proof outlines, before Section 2.4 describes the proof method.

For technical convenience, we first formulate verification conditions as standard Hoare-triples. The statements of these Hoare-triples may also contain assignments involving qualified references as given by the global assertion language. The formal semantics is given in Chapter 6 by means of a weakest precondition calculus [dB99].

Proof outlines For a complete proof system it is necessary that the transition semantics of $Java_{seq}$ can be encoded in the assertion language. As the assertion language reasons about the local and global states, we have to *augment* the program with fresh *auxiliary variables* to represent information about the control points and stack structures within the local and global states. Invariant program properties are specified by the *annotation.* An augmented and annotated program is called a *proof outline* or an *asserted program.*

Augmentation An augmentation extends a program by atomically executed multiple assignments $\vec{y} := \vec{e}$ to distinct auxiliary variables, which we call *observations*. Furthermore, the observations have, in general, to be "attached" to statements they observe in an atomic manner. For object creation this is syntactically represented by the augmentation $u := \mathsf{new}^c \; \langle \vec{y} := \vec{e} \rangle^{new}$ which attaches the observation to the object creation statement. Observations $\vec{y}_1 := \vec{e}_1$ of a method call and observations $\vec{y}_4 := \vec{e}_4$ of the corresponding reception of a return value are denoted by $u := e_0.m(\vec{e}) \; \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret}$. The augmentation $\langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} stm; \mathsf{return}\; e_{ret} \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret}$ of method bodies specifies $\vec{y}_2 := \vec{e}_2$ as the observation of the reception of the method call and $\vec{y}_3 := \vec{e}_3$ as the observation attached to the return statement. Assignments can be observed using $\vec{y} := \vec{e} \; \langle \vec{y}' := \vec{e}' \rangle^{ass}$. A stand-alone observation not attached to any statement is written as $\langle \vec{y} := \vec{e} \rangle$. It can be inserted at any point in the program.

Note that we could also use the same syntax for all kinds of observations. However, such a notation would be disadvantageous for partial augmentations, i.e., for the specification of augmentations where not all statements are observed. For example, using the notation introduced above, the augmentation $e_0.m(\vec{e}) \; \langle stm \rangle$ uniquely specifies stm as an alone-standing observation following an unobserved method call; using the same augmentation syntax $\langle stm \rangle$ for all kinds of observations, we would have to write $e_0.m(\vec{e}) \; \langle\rangle \, \langle\rangle \, \langle stm \rangle$ to specify the same setting. The same remark can be made also for the annotation syntax, introduced below.

The augmentation does not influence the control flow of the program but enforce a particular scheduling policy. An assignment statement and its observation are executed simultaneously. Object creation and its observation are executed in a single computation step, in this order. For method call, communication, sender, and receiver observations are executed in a single computation step, in this order (see Figure 2 on page 25 and Figure 3 of page 26). Points between a statement and its observation are no *control points*, since they are executed in a single computation step; we call them *auxiliary points*.

To exclude the possibility, that two multiple assignments get executed in a single computation step in the same object, we require that the caller observation in a self-communication may not change the values of instance variables. Without this restriction, we would have to show interference freedom under assignment-pairs, which would increase the complexity of the proof system. Formally, in each observation of a method invocation statement $e_0.m(\vec{e})$, assignments to instance variables must have the form $x := \mathsf{if}\;\; e_0 = \mathsf{this}\,\mathsf{then}\;\, x \;\mathsf{else}\; e\;\mathsf{fi}$.

In the following we call assignment statements with their observations, unobserved assignments, alone-standing observations, or observations of communication or object creation general as multiple assignments, since they are executed simultaneously.

Example 1. Extending an assignment $x := e$ to $x := e \; \langle u := x \rangle^{ass}$ stores the value of x *prior* to the execution of $x := e$ in the auxiliary variable u. Extending it to $x := e \; \langle u := x \rangle$ stores the value of x in u *after* the execution of $x := e$.

Example 2. We can store the number of objects created by an instance of a class c using an auxiliary integer instance variable n with initial value 0, and extending each object creation statement $u := \mathsf{new}^{c'}$ in c to $u := \mathsf{new}^{c'}\ \langle n := n+1\rangle^{new}$.

Example 3. We extend Example 2 by additionally observing each call $u := e_0.m(\vec{e})$ in c by $u := e_0.m(\vec{e})\ \langle k := n\rangle^{!call}\langle k := n-k\rangle^{?ret}$. Then the value of the auxiliary local integer variable k after method call, but before return stores the number of objects created up to the call. After return, it stores the number of objects created during method evaluation.

Example 4. Let l be an auxiliary integer instance variable of a class c. We can count the number of local configurations executing in an instance of c by augmenting the body $stm; \mathsf{return}\ e_{ret}$ of each method in class c resulting in $\langle l := l+1\rangle^{?call}\ stm;\ \mathsf{return}\ e_{ret}\ \langle l := l-1\rangle^{!ret}$.

The above examples show how to count objects, local configurations in an object, etc. But this information is not sufficient for a complete proof system: we have to be able to *identify* those entities. We identify a local configuration by the object in which it executes together with the value of its built-in auxiliary local variable conf storing a unique object-internal identifier. Its uniqueness is assured by the auxiliary instance variable counter, incremented for each new local configuration in that object. The callee receives the "return address" as auxiliary formal parameter caller of type Object $\times$ Int, storing the identities of the caller object and the calling local configuration. The run-method of the initial object is executed with the parameter caller having the value $(null, 0)$.

Syntactically, each method declaration $m(\vec{u})\{stm; \mathsf{return}\ e_{ret}\}$ gets extended by the built-in augmentation to $m(\vec{u}, \mathsf{caller})\{\langle \mathsf{conf}, \mathsf{counter} := \mathsf{counter}, \mathsf{counter} + 1\rangle^{?call} stm; \mathsf{return}\ e_{ret}\}$. Correspondingly for method calls $u := e_0.m(\vec{e})$, the actual parameter lists get extended resulting in $u := e_0.m(\vec{e}, (\mathsf{this}, \mathsf{conf}))$. The values of the built-in auxiliary variables must not be changed by the user-defined augmentation but may be used in the augmentation and annotation. In the examples of the following sections we don't list the built-in augmentation; they are meant to be automatically included in all proof outlines.

Annotation To specify invariant properties of the system, the augmented programs are *annotated* by attaching local assertions to each control and auxiliary point. We use the triple notation $\{p\}\, stm\, \{q\}$ and write $pre(stm)$ and $post(stm)$ to refer to the pre- and the post-condition of a statement. For assertions at auxiliary points we use the following notation: The annotation

$$\{p_0\}\ u := \mathsf{new}^c\ \{p_1\}^{new}\ \langle \vec{y} := \vec{e}\rangle^{new}\ \{p_2\}$$

of an object creation statement specifies p_0 and p_2 as pre- and postconditions, where p_1 at the auxiliary point should hold directly after object creation but before its observation. The annotation

$$\{p_0\}\, u := e_0.m(\vec{e}) \quad \{p_1\}^{!call}\langle \vec{y}_1 := \vec{e}_1\rangle^{!call} \quad \{p_2\}^{wait} \quad \{p_3\}^{?ret}\langle \vec{y}_4 := \vec{e}_4\rangle^{?ret} \quad \{p_4\}$$

assigns p_0 and p_4 as pre- and postconditions to the method invocation; p_1 is assumed to hold directly after method call, but prior to its observation; p_2 describes the control point of the caller after method call and before return; finally, p_3 specifies the state directly after return but before its observation. The annotation of method bodies $stm;\mathsf{return}\ e_{ret}$ is as follows:

$$\{p_0\}^{?call}\ \langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\ \{p_1\}\quad stm;\quad \{p_2\}\ \mathsf{return}\ e_{ret}\ \{p_3\}^{!ret}\ \langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\ \{p_4\}$$

The callee postcondition of the method call is p_1; the callee pre- and postconditions of return are p_2 and p_4. The assertions p_0 respectively p_3 specify the states of the callee between method call respectively return and its observation.

Besides pre- and postconditions, for each class c, the annotation defines a local assertion I_c called *class invariant*, specifying invariant properties of instances of c in terms of its instance variables.[6] We require that for each method of a class, the class invariant is the precondition of the method body.

Finally, a global assertion GI called the *global invariant* specifies properties of communication between objects. As such, it should be invariant under object-internal computation. For that reason, we require that for all qualified references $E.x$ in GI with E of type c, all assignments to x in class c occur in the observations of communication or object creation. We require furthermore that in the annotation no free logical variables occur. In the following we will use also partially annotated statements; assertions which are not explicitly specified are by definition true.

Example 5. The (partial) annotation $u := \mathsf{new}^c\ \{u \neq \mathsf{this}\}$ of an object creation statement in a class c' expresses that the new object's identity differs from the identity of the creator object. This annotation is invariant, independently of the rest of the program, since the new object's identity is fresh and the only shared variable in the assertion is the self-reference, which may not be assigned to.

The same property can be expressed using the class invariant. Since the class invariant may refer to instance variables only, we have to store the new object's identity in an auxiliary instance variable x in order to refer to it in the class invariant. We define the annotation $u := \mathsf{new}^c\ \langle x := u\rangle^{new}\{x = u\}$ and the class invariant by $x \neq \mathsf{this}$. In this case, invariance of the given assertions depends also on the rest of the class definition: an observation $x := \mathsf{this}$ executed in the same object would of course heart the class invariant. This annotation is useful, if different assertions in the same class refer to x, and especially if the information expressed by the class invariant is needed to show properties of incoming method calls.

Also the global invariant can be used to express the above property: Assume again $u := \mathsf{new}^c\ \langle x := u\rangle^{new}\{x = u\}$ and let the global invariant be defined by $\forall(z : c').\, z.x \neq z$. Again, the invariance of the annotation depends on the

[6] The notion of class invariant commonly used for sequential object-oriented languages differs from our notion: In a sequential setting, it would be sufficient that the class invariant holds initially and is preserved by whole method calls, but not necessarily in between.

rest of the class. But now it additionally depends also on the definition of other classes, possibly creating new instances of c', thereby extending the domain of the quantification. Such annotations are used to express dependencies between different instance states.

Verification conditions The proof system formalizes a number of *verification conditions* which inductively ensure that for each reachable configuration the local assertions attached to the current control points in the thread configuration as well as the global and the class invariants hold. The conditions are grouped, as usual, into initial conditions, and for the inductive step into local correctness and tests for interference freedom and cooperation.

Before specifying the verification conditions, we first list some notation. Let Init be a syntactical operator with interpretation *Init* (cf. page 9). Given $IVar_c$ as the set of instance variables of class c without the self-reference, and z a logical variable of type c, let $\mathsf{InitState}(z)$ be the global assertion $z \neq \mathsf{null} \wedge \bigwedge_{x \in IVar_c} z.x = \mathsf{Init}(x)$, expressing that the object denoted by z is in its initial instance state.

Finally, arguing about two different local configurations makes it necessary to distinguish between their local variables, since they may have the same names; in such cases we will rename the local variables in one of the local states. We use primed assertions p' to denote the given assertion p with every local variable u replaced by a fresh one u', and correspondingly for expressions.

Initial correctness A proof outline is *initially correct*, if the precondition of the main statement, the class invariant of the initial object, and the global invariant are satisfied initially, i.e., in the initial global configuration after the execution of the callee observation at the beginning of the main statement. Furthermore, the precondition of the observation should be satisfied prior to its execution.

Definition 1 (Initial correctness). *Let the body of the* run*-method of the main class c be* $\{p_2\}^{?call}\langle\vec{y}_2 := \vec{e}_2\rangle^{?call}\{p_3\}\ stm; \mathsf{return}$ *with local variables $\vec{v}$ without the formal parameters, $z \in LVar^c$, and $z' \in LVar^{\mathsf{Object}}$. A proof outline is* initially correct, *if*

$$\models_{\mathcal{G}} \quad \{\mathsf{InitState}(z) \wedge \forall z'.\ z' = \mathsf{null} \vee z = z'\} \qquad (1)$$
$$\vec{v}, \mathsf{caller} := \mathsf{Init}(\vec{v}), (\mathsf{null}, 0)$$
$$\{P_2(z)\}$$
$$\models_{\mathcal{G}} \quad \{\mathsf{InitState}(z) \wedge \forall z'.\ z' = \mathsf{null} \vee z = z'\} \qquad (2)$$
$$\vec{v}, \mathsf{caller} := \mathsf{Init}(\vec{v}), (\mathsf{null}, 0); \quad z.\vec{y}_2 := \vec{E}_2(z)$$
$$\{GI \wedge P_3(z) \wedge I_c(z)\}$$

The assertion $\mathsf{InitState}(z) \wedge \forall z'.\ z' = \mathsf{null} \vee z = z'$ states that the initial global state defines exactly one existing object z being in its initial instance state. Initialization of the local configuration is represented by the assignment $\vec{v}, \mathsf{caller} := \mathsf{Init}(\vec{v}), (\mathsf{null}, 0)$. The observation $\vec{y}_2 := \vec{e}_2$ at the beginning of the run-method of the initial object z is represented by the assignment $z.\vec{y}_2 := \vec{E}_2(z)$.

Example 6. Assume the following proof outline:

```
{∃(z1 : Initial). z1 ≠ null ∧ ∀(z2 : Initial). z2 ≠ null → z1 = z2}  //global invariant

class Initial{
    Int x;

    {started}  //class invariant

    Void run(){
        Int v;
        ⟨Int u; ⟩

        {u = 0 ∧ v = 0 ∧ x = 0}?call  //precondition of observation
        ⟨u := 1⟩?call                 //observation of call
        {u = 1 ∧ v = 0 ∧ x = 0}       //postcondition of observation
        ...
    }
}
```

Note that the built-in augmentation extends the observation $\{u := 1\}^{?call}$ to $\{u, \mathsf{started} := 1, \mathsf{true}\}^{?call}$. The first initial condition

$$\begin{gathered}\models_{\mathcal{G}} \{z \neq \mathsf{null} \wedge z.x = 0 \wedge \forall(z' : \mathsf{Object}).\ z' = \mathsf{null} \vee z = z'\} \\ v, u, \mathsf{caller} := 0, 0, (\mathsf{null}, 0) \\ \{u = 0 \wedge v = 0 \wedge z.x = 0\}\end{gathered}$$

assures that the precondition of the observation holds after initialization but prior to its execution. The second condition

$$\begin{gathered}\models_{\mathcal{G}} \{z \neq \mathsf{null} \wedge z.x = 0 \wedge \forall(z' : \mathsf{Object}).\ z' = \mathsf{null} \vee z = z'\} \\ v, u, \mathsf{caller} := 0, 0, (\mathsf{null}, 0); \quad u, z.\mathsf{started} := 1, \mathsf{true} \\ \{\mathit{GI} \wedge (u = 1 \wedge v = 0 \wedge x = 0) \wedge (z.\mathsf{started})\}\end{gathered}$$

assures that the global invariant, the postcondition of the observation, and the class invariant hold after the observation. Satisfaction of the global invariant can be shown by instantiation with z.

Local correctness A proof outline is *locally correct*, if the properties of method instances as specified by the annotation are invariant under their own execution, i.e., if the usual verification conditions [Apt81] for standard sequential constructs hold. For example, the precondition of an assignment must imply its postcondition after its execution. The following condition should hold for all multiple assignments being an assignment statement with its observation, an unobserved assignment, or an alone-standing observation:

Definition 2 (Local correctness: Assignment). *A proof outline is* locally correct*, if for all multiple assignments* $\{p_1\}\, \vec{y} := \vec{e}\, \{p_2\}$ *in class* c*, which is not the observation of object creation or communication,*

$$\models_{\mathcal{L}} \{p_1\} \quad \vec{y} := \vec{e} \quad \{p_2\}\ . \tag{3}$$

The conditions for loops and conditional statements are similar. Note that we have no local verification conditions for observations of communication and object creation. The postconditions of such statements express *assumptions* about the communicated values. These assumptions will be verified in the *cooperation test.*

Example 7. Assume the following augmented and annotated method which computes the faculty $u!$ for its parameter u:

```
Int fac(Int u){
    Int result;
    {u > 0}
    result:=1;  {result = 1 ∧ u > 0}
    v:=u;  {u! = result * v! ∧ u > 0 ∧ v > 0}
    while (v>1) do  {u! = result * v! ∧ u > 0 ∧ v > 1}
        result:=result*v;  {u! = result * (v − 1)! ∧ u > 0 ∧ v > 1}
        v:=v-1;  {u! = result * v! ∧ u > 0 ∧ v > 0}
    od;   {u! = result}
    return result
}
```

The above proof outline satisfies the conditions of local correctness. There are 7 local correctnes conditions (there are no initial correctness, interference freedom, and cooperation test conditions for this example). For example, for the assignment $\mathsf{result} := \mathsf{result} * v$ local correctness defines the verification condition

$$\models_{\mathcal{L}} \qquad \{u! = \mathsf{result} * v! \wedge u > 0 \wedge v > 1\}$$
$$\mathsf{result} := \mathsf{result} * v \quad \{u! = \mathsf{result} * (v-1)! \wedge u > 0 \wedge v > 1\}\ ,$$

whose satisfaction is easy to see.

The interference freedom test Invariance of local assertions under computation steps in which they are not involved is assured by the proof obligations of the *interference freedom test.* Its definition covers also invariance of the class invariants. Since $Java_{seq}$ does not support qualified references to instance variables, we only have to deal with invariance under execution within the *same* object. Affecting only local variables, communication and object creation do not change the instance states of the executing objects. Thus we only have to cover invariance of assertions at control points over assignments, including observations of communication and object creation. To distinguish local variables of the different local configurations, we rename those of the assertion.

Let q be an assertion at a control point and $\vec{y} := \vec{e}$ a multiple assignment in the same class c. In which cases does q have to be invariant under the execution of the assignment? Since the language is sequential, i.e., q and $\vec{y} := \vec{e}$ belong to the *same* thread, the only assertions endangered are those at control points waiting for return earlier in the current execution stack. Invariance of a local configuration under its own execution, however, need not be considered and is excluded by requiring $\mathsf{conf} \neq \mathsf{conf}'$. Interference with the *matching* return statement in a self-communication need also not be considered, because communicating partners execute simultaneously. Let $\mathsf{caller\_obj}$ be the first and $\mathsf{caller\_conf}$ the second component of caller. We define $\mathsf{waits\_for\_ret}(q, \vec{y} := \vec{e})$ by

- $\mathsf{conf}' \neq \mathsf{conf}$, for assertions $\{q\}^{wait}$ attached to control points waiting for return, if $\vec{y} := \vec{e}$ is not the observation of return;

– $\mathsf{conf}' \neq \mathsf{conf} \wedge (\mathsf{this} \neq \mathsf{caller\_obj} \vee \mathsf{conf}' \neq \mathsf{caller\_conf})$, for assertions $\{q\}^{wait}$, if $\vec{y} := \vec{e}$ observes return;
– false, otherwise.

For the example configuration intuitively shown in Fig. 1, the assertion p_3 attached to a control point waiting for return, has to be invariant under the execution of the assignment by its callee, while p_4 does not have to be invariant under its own execution. However, if the assignment would observe returning, then p_3 would not have to be invariant under the assignment. The assertions p_1 and p_2 are automatically invariant, since they describe an object different from the executing one. Note that satisfaction of p_5 after execution is assured by the local correctness conditions.

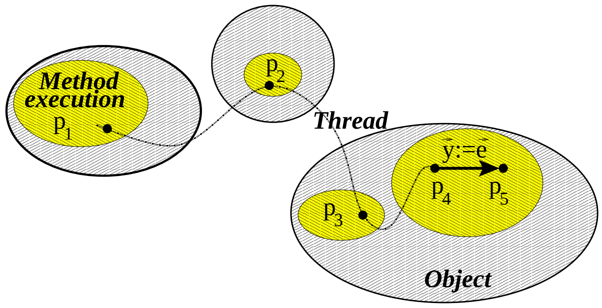


Fig. 1. Interference for a single thread

The interference freedom test can now be formulated as follows:

Definition 3 (Interference freedom). *A proof outline is* interference free, *if for all classes c and multiple assignments $\vec{y} := \vec{e}$ with precondition p in c,*

$$\models_{\mathcal{L}} \{p \wedge I_c\} \quad \vec{y} := \vec{e} \quad \{I_c\} . \tag{4}$$

Furthermore, for all assertions q at control points in c,

$$\models_{\mathcal{L}} \{p \wedge q' \wedge \mathsf{waits\_for\_ret}(q, \vec{y} := \vec{e})\} \quad \vec{y} := \vec{e} \quad \{q'\} . \tag{5}$$

Note that if we would allow qualified references in program expressions, we would have to show interference freedom of all assertions under all assignments in programs, not only for those occurring in the same class. For a program with n classes where each class contains k assignments and l assertions at control points, the number of interference freedom conditions is in $\mathcal{O}(c \cdot k \cdot l)$, instead of $\mathcal{O}((c \cdot k) \cdot (c \cdot l))$ with qualified references.

Example 8. Let $\{p_1\}\,\mathsf{this}.m(\vec{e})\ \{p_2\}^{!call}\langle stm_1\rangle^{!call}\{p_3\}^{wait}\{p_4\}^{?ret}\langle stm_2\rangle^{?ret}\{p_5\}$ be an annotated method call statement in a method m' of a class c with an integer auxiliary instance variable x, such that all assertions imply $\mathsf{conf} = x$. I.e., the identity of the executing local configuration is stored in the instance variable x. The annotation expresses that the method m' of c is not called recursively. That means, no pairs of control points in m' of c can be simultaneously reached.

The assertions p_2 and p_4 do not have to be shown invariant, since they are attached to auxiliary points. Interference freedom neither requires invariance of the assertions p_1 and p_5, since they are not at control points waiting for return, and thus the antecedents of the corresponding conditions evaluate to false. Invariance of p_3 under the execution of the observation stm_1 with precondition p_2 requires validity of $\models_{\mathcal{L}} \{p_2 \wedge p_3' \wedge \mathsf{waits\_for\_ret}(p_3, stm_1)\}\ stm_1\ \{p_3'\}$. The assertion $p_2 \wedge p_3' \wedge \mathsf{waits\_for\_ret}(p_3, stm_1)$ implies $(\mathsf{conf} = x) \wedge (\mathsf{conf}' = x) \wedge (\mathsf{conf}' \neq \mathsf{conf})$, which evaluates to false. Invariance of p_3 under stm_2 is analogous.

Example 9. Assume a partially[7] annotated method invocation statement of the form $\{p_1\}\,\mathsf{this}.m(\vec{e})\ \{\mathsf{conf} = x \wedge p_2\}^{wait}\{p_3\}$ in a class c with an integer auxiliary instance variable x, and assume that method m of c has the annotated return statement $\{q_1\}\ \mathsf{return}\ \{\mathsf{caller} = (\mathsf{this}, x)\}^{!ret}\langle stm\rangle^{!ret}\{q_2\}$. The annotation expresses that the local configurations containing the above statements are in caller-callee relationship. Thus upon return, the control point of the caller moves from the point at $\mathsf{conf} = x \wedge p_2$ to that at p_3, i.e, $\mathsf{conf} = x \wedge p_2$ does not have to be invariant under the observation of the return statement.

Again, the assertion $\mathsf{caller} = (\mathsf{this}, x)$ at an auxiliary point does not have to be shown invariant. For the assertions p_1, p_3, q_1, and q_2, which are not at a control point waiting for return, the antecedent is false. Invariance of $\mathsf{conf} = x \wedge p_2$ under the observation stm with precondition $\mathsf{caller} = (\mathsf{this}, x)$ is covered by the interference freedom condition

$$\begin{aligned}\models_{\mathcal{L}} \{\ & \mathsf{caller} = (\mathsf{this}, x) \wedge (\mathsf{conf}' = x \wedge p_2') \wedge \\ & \mathsf{waits\_for\_ret}((\mathsf{conf} = x \wedge p_2), stm) \quad \}\ stm\ \{\mathsf{conf}' = x \wedge p_2'\}\ .\end{aligned}$$

The $\mathsf{waits\_for\_ret}$ assertion implies $\mathsf{caller} \neq (\mathsf{this}, \mathsf{conf}')$, which contradicts the assumptions $\mathsf{caller} = (\mathsf{this}, x)$ and $\mathsf{conf}' = x$; thus the antecedent of the condition is false.

Satisfaction of $\mathsf{caller} = (\mathsf{this}, x)$ directly after communication and satisfaction of p_3 and q_2 after the observation is assured by the cooperation test.

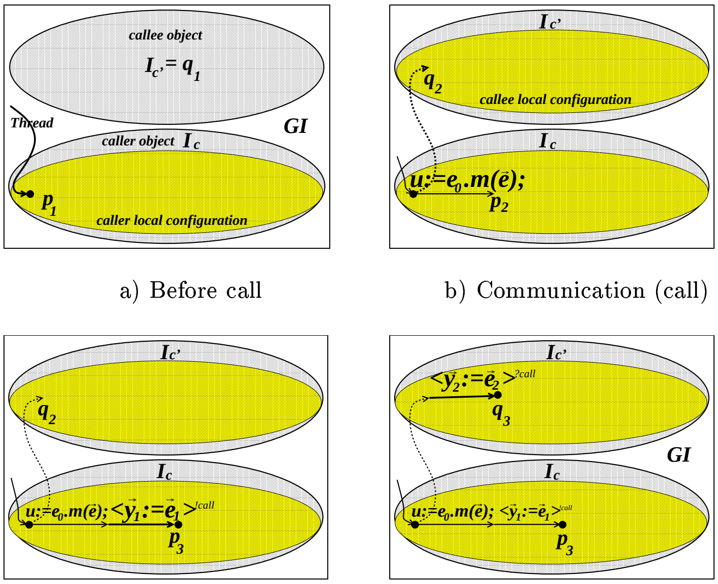
The cooperation test Whereas the interference freedom test assures invariance of assertions under steps in which they are not involved, the *cooperation test* deals with inductivity for communicating partners, assuring that the global invariant and the preconditions of the involved statements imply their postconditions after the joint step. Additionally, the preconditions of the corresponding observations must hold immediately after communication.

The global invariant refers to auxiliary instance variables which are allowed to be changed by observations of communication, only. Consequently, the global invariant is automatically invariant under the execution of non-communicating

[7] As already mentioned, missing assertions are by definition true.

statements. For communication and object creation, however, the invariance must be shown as part of the cooperation test.

We start with the cooperation test for method invocation. The semantics of method call and returning from a method is intuitively shown in Figures 2 and 3. After communication, i.e., after creating and initializing the callee local configuration and passing on the actual parameters, first the caller, and then the callee execute their corresponding observations, all in a single computation step. Correspondingly for return, after communicating the result value, first the callee and then the caller observation gets executed. Since different objects may



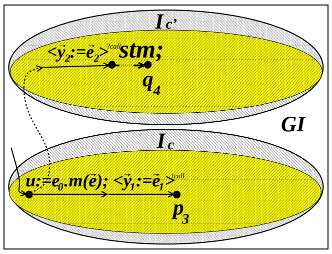
a) Before call — b) Communication (call)

c) Caller observation — d) Callee observation

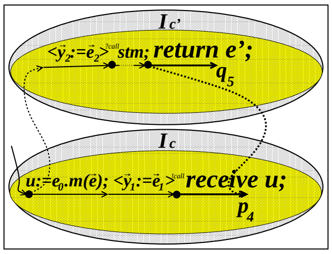
Fig. 2. Execution of a method call $\{p_1\}\, u := e_0.m(\vec{e})\ \{p_2\}^{!call}\langle \vec{y}_1 := \vec{e}_1\rangle^{!call}\{p_3\}^{wait}$ with callee method body $\{q_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{q_3\}\ stm; \mathsf{return}\ e'$.
Control points are marked by a circle.

be involved, the cooperation test is formulated in the global assertion language. Local properties are expressed in the global language using the lifting substitution. As already mentioned, we use the shortcuts $P(z)$ for $p[z/\mathsf{this}]$, $Q'(z')$ for $q'[z'/\mathsf{this}]$, and similarly for expressions. To avoid name clashes between local variables of the partners, we rename those of the callee.

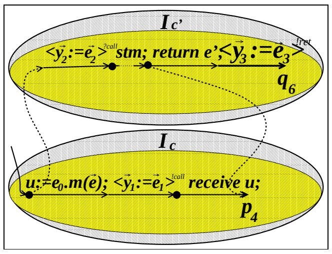
Let z and z' be logical variables representing the caller, respectively the callee object in a method call. We assume the global invariant and the preconditions of the communicating statements to hold prior to communication. For method



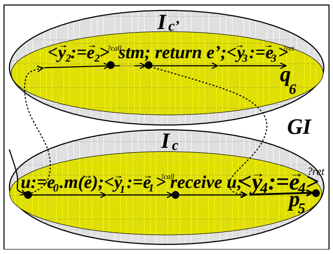
e) Method evaluation



f) Communication (return)



g) Callee observation



h) Caller observation

Fig. 3. Execution of return for a method call
$\{p_1\}\, u := e_0.m(\vec{e})\ \{p_2\}^{!call} \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \{p_3\}^{wait} \{p_4\}^{?ret} \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} \{p_5\}$
with callee method body
$\{q_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{q_3\}\ stm; \{q_4\}\ \mathsf{return}\ e'\ \{q_5\}^{!ret} \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret} \{q_6\}$.
Control points are marked by a circle.

invocation, the precondition of the callee is its class invariant. That the two statements indeed represent communicating partners is captured in the assertion comm, which depends on the type of communication: For method invocation $e_0.m(\vec{e})$, the assertion $E_0(z) = z'$ states, that z' is indeed the callee object. Remember that method invocation hands over the return address, and that the values of formal parameters remain unchanged. Furthermore, actual parameters may not contain instance variables, i.e., their interpretation does not change during method execution. Therefore, the formal and actual parameters can be used at returning from a method to identify partners being in caller-callee relationship, using the built-in auxiliary variables. Thus for the return case, comm additionally states $\vec{u}' = \vec{E}(z)$, where $\vec{u}$ and $\vec{e}$ are the formal and the actual parameters. Returning from the run-method terminates the executing thread, which does not have communication effects.

As in the previous conditions, state changes are represented by assignments. For the example of method invocation, communication is represented by the assignment $\vec{u}' := \vec{E}(z)$, where initialization of the remaining local variables $\vec{v}$ is covered by $\vec{v}' := \mathsf{Init}(\vec{v})$. The assignments $z.\vec{y}_1 := \vec{E}_1(z)$ and $z'.\vec{y}'_2 := \vec{E}'_2(z')$

stand for the caller and callee observations $\vec{y}_1 := \vec{e}_1$ and $\vec{y}_2 := \vec{e}_2$, executed in the objects z and z', respectively. Note that we rename all local variables of the callee to avoid name clashes.

Definition 4 (Cooperation test: Communication). *A proof outline satisfies the* cooperation test for communication, *if*

$$\begin{aligned}\models_{\mathcal{G}} \{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\}\\ f_{comm}\\ \{P_2(z) \wedge Q_2'(z')\}\end{aligned} \tag{6}$$

$$\begin{aligned}\models_{\mathcal{G}} \{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\}\\ f_{comm};\quad f_{obs1};\quad f_{obs2}\\ \{GI \wedge P_3(z) \wedge Q_3'(z')\}\end{aligned} \tag{7}$$

holds for distinct fresh logical variables $z \in LVar^c$ and $z' \in LVar^{c'}$, in the following cases:

1. Call*: For all statements $\{p_1\}\, u_{ret} := e_0.m(\vec{e})\ \{p_2\}^{!call}\langle\vec{y}_1 := \vec{e}_1\rangle^{!call}\{p_3\}^{wait}$ (or such without receiving a value) in class c with e_0 of type c', where method m of c' has body $\{q_2\}^{?call}\langle\vec{y}_2 := \vec{e}_2\rangle^{?call}\{q_3\}\ stm;\mathsf{return}\ e_{ret}$, formal parameters $\vec{u}$, and local variables $\vec{v}$ except the formal parameters. The callee class invariant is $q_1 = I_{c'}$. The assertion* comm *is given by $E_0(z) = z'$. Furthermore, f_{comm} is $\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v})$, f_{obs1} is $z.\vec{y}_1 := \vec{E}_1(z)$, and f_{obs2} is $z'.\vec{y}_2' := \vec{E}_2'(z')$.*
2. Return*: For all $u_{ret} := e_0.m(\vec{e})\ \langle stm\rangle^{!call}\{p_1\}^{wait}\{p_2\}^{?ret}\langle\vec{y}_4 := \vec{e}_4\rangle^{?ret}\{p_3\}$ (or such without receiving a value) occurring in c with e_0 of type c', such that method m of c' has the return statement $\{q_1\}\ \mathsf{return}\ e_{ret}\ \{q_2\}^{!ret}\langle\vec{y}_3 := \vec{e}_3\rangle^{!ret}\{q_3\}$, and formal parameter list $\vec{u}$, the above equations must hold with* comm *given by $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$, and where f_{comm} is $u_{ret} := E'_{ret}(z')$, f_{obs1} is $z'.\vec{y}_3' := \vec{E}_3'(z')$, and f_{obs2} is $z.\vec{y}_4 := \vec{E}_4(z)$.*
3. Return$_{run}$*: For $\{q_1\}$* return *$\{q_2\}^{!ret}\langle\vec{y}_3 := \vec{e}_3\rangle^{!ret}\{q_3\}$ occurring in the* run*-method of the main class, $p_1 = p_2 = p_3 =$* true, comm = true*, and furthermore f_{comm} and f_{obs2} are the empty statement, and f_{obs1} is $z'.\vec{y}_3' := \vec{E}_3'(z')$.*

Example 10. This example illustrates how one can prove properties of parameter passing. Let $\{p\}\, e_0.m(v, \vec{e})$, with p given by $v > 0$, be a (partially) annotated statement in a class c with e_0 of type c', and let method $m(u, \vec{w})$ of c' have the body $\{q\}\ stm;\mathsf{return}$ where q is $u > 0$. Inductivity of the proof outline requires that if p is valid prior to the call (besides the global and class invariants), then q is satisfied after the invocation. Omitting irrelevant details, Condition 7 of the cooperation test requires proving $\models_{\mathcal{G}} \{P(z)\}\ u' := v\ \{Q'(z')\}$, which expands to $\models_{\mathcal{G}} \{v > 0\}\ u' := v\ \{u' > 0\}$.

Example 11. The following example demonstrates how one can express dependencies between instance states in the global invariant and use this information in the cooperation test.

Let $\{p\}\, e_0.m(\vec{e})$, with p given by $x > 0 \wedge e_0 = o$, be an annotated statement in a class c with e_0 of type c', x an integer instance variable, and o an instance variable of type c', and let method $m(\vec{u})$ of c' have the annotated body $\{q\}\, stm; \mathsf{return}$ where q is $y > 0$ and y an integer instance variable. Let furthermore $z \in LVar^c$ and let the global invariant be given by $\forall z. (z \neq \mathsf{null} \wedge z.o \neq \mathsf{null} \wedge z.x > 0) \to z.o.y > 0$. Inductivity requires that if p and the global invariant are valid prior to the call, then q is satisfied after the invocation (again, we omit irrelevant details). The cooperation test Condition 7, i.e., $\models_{\mathcal{G}} \{GI \wedge P(z) \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\}\ \vec{u}' := \vec{E}(z)\ \{Q'(z')\}$ expands to

$$\begin{array}{c}
\models_{\mathcal{G}} \{(\forall z. (z \neq \mathsf{null} \wedge z.o \neq \mathsf{null} \wedge z.x > 0) \to z.o.y > 0) \wedge \\
(z.x > 0 \wedge E_0(z) = z.o) \wedge E_0(z) = z' \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\ \} \\
\vec{u}' := \vec{E}(z) \\
\{z'.y > 0\}
\end{array}$$

Instantiating the quantification by z, the antecedent implies $z.o.y > 0 \wedge z' = z.o$, i.e., $z'.y > 0$. Invariance of the global invariant is straightforward.

Example 12. This example illustrates how the cooperation test handles observations of communication. Let $\{\neg b\}\, \mathsf{this}.m(\vec{e})\{b\}^{wait}$ be an annotated statement in a class c with boolean auxiliary instance variable b and let $m(\vec{u})$ of c have the body $\{\neg b\}^{?call} \langle b := \mathsf{true}\rangle^{?call} \{b\}\, stm; \mathsf{return}$. Condition 6 of the cooperation test assures inductivity for the precondition of the observation. We have to show $\models_{\mathcal{G}} \{\neg z.b \wedge \mathsf{comm}\}\vec{u}' := \vec{E}(z)\{\neg z'.b\}$, i.e., since it is a self-call, $\models_{\mathcal{G}} \{\neg z.b \wedge z = z'\}\vec{u}' := \vec{E}(z)\{\neg z'.b\}$, which is trivially satisfied. Condition 7 of the cooperation test for the postconditions requires $\models_{\mathcal{G}} \{\mathsf{comm}\}\vec{u}' := \vec{E}(z); z'.b := \mathsf{true}\{z.b \wedge z'.b\}$ which expands to $\models_{\mathcal{G}} \{z = z'\}\vec{u}' := \vec{E}(z); z'.b := \mathsf{true}\{z.b \wedge z'.b\}$, whose validity is easy to see.

Besides method calls and returns, the cooperation test needs to handle object creation, taking care of the preservation of the global invariant, the postcondition of the new statement and its observation, and the new object's class invariant. We can assume that the precondition of the object creation statement and the global invariant hold in the configuration prior to instantiation. The extension of the global state with a freshly created object is formulated in a *strongest postcondition* style, i.e., it is required to hold immediately *after* the instantiation. We use existential quantification to refer to the old value: z' of type $LVar^{\mathsf{list\,Object}}$ represents the existing objects prior to the extension. Moreover, that the created object's identity stored in u is fresh and that the new instance is properly initialized is expressed by the global assertion $\mathsf{Fresh}(z', u)$ defined as $\mathsf{InitState}(u) \wedge u \notin z' \wedge \forall v.\ v \in z' \vee v = u$ (see page 20 for the definition of $\mathsf{InitState}$). To express that an assertion refers to the set of existing objects *prior* to the extension of the global state, we need to *restrict* any existential quantification in the assertion to range over objects from z', only. So let P be a global assertion and $z' \in LVar^{\mathsf{list\,Object}}$ a logical variable not occurring in P.

Then $P \downarrow z'$ is the global assertion P with all quantifications $\exists z.\ P'$ replaced by $\exists z.\ \mathsf{obj}(z) \subseteq z' \wedge P'$, where $obj(v)$ denotes the set of objects occurring in the value v. The following lemma formulates the basic property of the projection operator:

Lemma 2. *Assume a global state σ, an extension $\sigma' = \sigma[\alpha \mapsto \sigma_{inst}^{c,init}]$ for some $\alpha \in Val^c$, $\alpha \notin Val(\sigma)$, and a logical environment ω referring only to values existing in σ. Let v be the sequence consisting of all elements of $\bigcup_c Val_{null}^c(\sigma)$. Then for all global assertions P and logical variables $z' \in LVar^{\mathsf{list\,Object}}$ not occurring in P,*

$$\omega, \sigma \models_{\mathcal{G}} P \quad \textit{iff} \quad \omega[z' \mapsto v], \sigma' \models_{\mathcal{G}} P \downarrow z'.$$

The proof can be found in Appendix A. Thus a predicate $(\exists u.\ P) \downarrow z'$, evaluated immediately after the instantiation, expresses that P holds prior to the creation of the new object. This leads to the following definition of the cooperation test for object creation:

Definition 5 (Cooperation test: Instantiation). *A proof outline satisfies the* cooperation test for object creation, *if for all classes c' and statements $\{p_1\}\, u := \mathsf{new}^c\ \{p_2\}^{new} \langle \vec{y} := \vec{e} \rangle^{new} \{p_3\}$ in c':*

$$\begin{aligned} \models_{\mathcal{G}} \quad & z{\neq}\mathsf{null} \wedge z{\neq}u \wedge \exists z'.\ \big(\mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u.\ P_1(z)) \downarrow z'\big) \\ & \qquad\qquad\qquad \rightarrow P_2(z) \wedge I_c(u) \end{aligned} \tag{8}$$

$$\begin{aligned} \models_{\mathcal{G}} \quad & \{z{\neq}\mathsf{null} \wedge z{\neq}u \wedge \exists z'.\ \big(\mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u.\ P_1(z)) \downarrow z'\big)\} \\ & \qquad\qquad\qquad z.\vec{y} := \vec{E}(z) \\ & \qquad\qquad\qquad \{GI \wedge P_3(z)\} \end{aligned} \tag{9}$$

with $z \in LVar^{c'}$ and $z' \in LVar^{\mathsf{list\,Object}}$ fresh.

Example 13. Assume a statement $u := \mathsf{new}^c\{u \neq \mathsf{this}\}$ in a program, where the class invariant of c is $x \geq 0$ for an integer instance variable x. Condition 8 of the cooperation test for object creation assures that the class invariant of the new object holds after its creation. We have to show validity of $\models_{\mathcal{G}} (\exists z'.\ \mathsf{Fresh}(z', u)) \rightarrow u.x \geq 0$, i.e., $\models_{\mathcal{G}} u.x = 0 \rightarrow u.x \geq 0$, which is trivial. For the postcondition, Condition 9 requires $\models_{\mathcal{G}} \{z \neq u\}\epsilon\{u \neq z\}$ with ϵ the empty statement (no observations are executed), which is true.

3 The concurrent language

In this section we extend the language $Java_{seq}$ to a *concurrent* language $Java_{conc}$ by allowing *dynamic thread creation.* Again, we define syntax and semantics of the language, before formalizing the proof system for the concurrent language.

$$
\begin{aligned}
meth &::= m(u,\ldots,u)\{\ stm; \mathsf{return}\ exp_{ret}\} \\
meth_{\mathsf{run}} &::= \mathsf{run}()\{\ stm; \mathsf{return}\ \} \\
class &::= \mathsf{class}\ c\{meth\ldots meth\ meth_{\mathsf{run}}\ meth_{\mathsf{start}}\} \\
class_{\mathsf{main}} &::= class \\
prog &::= \langle class\ldots class\ class_{\mathsf{main}}\rangle
\end{aligned}
$$

Table 7. $Java_{conc}$ abstract syntax

3.1 Syntax

Expressions and statements can be constructed as in $Java_{seq}$. The abstract syntax of the remaining constructs is summarized in Table 7. As we focus on concurrency aspects, all classes are `Thread` classes in the sense of *Java*: Each class contains a pre-defined start-method that can be invoked only once for each object, resulting in a new thread of execution. The new thread starts to execute the user-defined run-method of the given object while the initiating thread continues its own execution. The run-methods cannot be invoked directly. The parameterless start-method without return value is not implemented syntactically; see the next section for its semantics. Note, that the syntax does not allow qualified references to instance variables. As a consequence, shared-variable concurrency is caused by simultaneous execution within a single object, only, but not across object boundaries.

3.2 Semantics

The operational semantics of $Java_{conc}$ extends the semantics of $Java_{seq}$ by dynamic thread creation. The additional rules are shown in Table 8. The invoca-

$$
\frac{\beta = [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in Val^{c}(\sigma) \qquad \neg started(T \cup \{\xi \circ (\alpha, \tau, e.\mathsf{start}(); stm)\}, \beta)}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, e.\mathsf{start}(); stm)\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, stm), (\beta, \tau_{init}^{\mathsf{run},c}, body_{\mathsf{run},c})\}, \sigma\rangle}\ \textsc{Call}_{start}
$$

$$
\frac{\beta = [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in Val(\sigma) \qquad started(T \cup \{\xi \circ (\alpha, \tau, e.\mathsf{start}(); stm)\}, \beta)}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, e.\mathsf{start}(); stm)\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, stm)\}, \sigma\rangle}\ \textsc{Call}_{start}^{skip}
$$

Table 8. $Java_{conc}$ operational semantics

tion of a start-method brings a new thread into being (rule $\textsc{Call}_{start}$). Only the first invocation of the start-method has this effect (rule $\textsc{Call}_{start}^{skip}$).[8] This

[8] In *Java* an exception is thrown if the thread is already started but not yet terminated.

is captured by the predicate $started(T, \beta)$ which holds iff there exists a stack $(\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n) \in T$ such that $\beta = \alpha_0$. A thread ends its lifespan by returning from a run-method (rule RETURN_{run} of Table 3).[9]

3.3 The proof system

In contrast to the sequential language, the proof system additionally has to accommodate for dynamic thread creation and shared-variable concurrency. Before describing the proof method, we show how to extend the built-in augmentation of the sequential language.

Proof outlines To get a complete proof system, for the concurrent language we additionally have to be able to identify *threads*. We identify a thread by the object in which it has begun its execution. We use the type Thread thus as abbreviation for the type Object. This identification is unique, since an object's thread can be started only once. During a method call, the callee thread receives its own identity as an auxiliary formal parameter thread. Additionally, we extend the auxiliary formal parameter caller by the caller thread identity, i.e., let caller be of type $\mathsf{Object} \times \mathsf{Int} \times \mathsf{Thread}$, storing the identities of the caller object, the calling local configuration, and the caller thread. Note that the thread identities of caller and callee are the same in all cases but the invocation of a start-method. The run-method of the initial object is executed with the parameters $(\mathsf{thread}, \mathsf{caller})$ having the values $(\alpha_0, (null, 0, null))$, where α_0 is the initial object. The boolean instance variable started, finally, remembers whether the object's start-method has already been invoked.

Syntactically, each formal parameter list $\vec{u}$ in the original program gets extended to $(\vec{u}, \mathsf{thread}, \mathsf{caller})$. Correspondingly for the caller, each actual parameter list $\vec{e}$ in statements invoking a method different from start gets extended to $(\vec{e}, \mathsf{thread}, (\mathsf{this}, \mathsf{conf}, \mathsf{thread}))$. The invocation of the parameterless start-method of an object e_0 gets the actual parameter list $(e_0, (\mathsf{this}, \mathsf{conf}, \mathsf{thread}))$. Finally, the callee observation at the beginning of the run-method executes $\mathsf{started} := \mathsf{true}$. The variables conf and counter are updated as in the previous section.

Remember that the caller observation of self-calls may not modify the instance state, as required in Section 2.4. Invoking the start-method by a self-call is specific in that, when the thread is already started, the caller is the only active entity. In this case, it has to be the caller that updates the instance state; the corresponding observation has the form $x := \mathsf{if}\ e_0 = \mathsf{this} \wedge \neg \mathsf{started}\ \mathsf{then}\ x\ \mathsf{else}\ e\ \mathsf{fi}$.

Since a thread calling a start method does not wait for return but continues execution, the augmentation and annotation of such method invocations have the form $\{p_1\}\ e_0.\mathsf{start}(\vec{e})\ \{p_2\}^{!call} \langle stm \rangle^{!call} \{p_3\}$.

[9] The worked-off local configuration (α, τ, ϵ) is kept in the global configuration to ensure that the thread of α cannot be started twice.

Verification conditions Initial correctness changes only, in that the formal parameters thread and caller get the initial values z and $(null, 0, null)$. Local correctness is not influenced by the new issue of concurrency. Note that local correctness applies now to all concurrently executing threads.

The interference freedom test Interference of a *single* thread under its own execution remains the same as for the sequential language. However, we additionally have to deal with invariance of properties of a thread under the execution of a *different* thread. Note that assertions at auxiliary points do not have to be shown invariant. Again, to distinguish local variables of the different local configurations, we rename those of the assertion which we show to be invariant.

An assertion q at a control point has to be invariant under an assignment $\vec{y} := \vec{e}$ in the same class only if the local configuration described by the assertion is not active in the computation step executing the assignment. If q and $\vec{y} := \vec{e}$ belong to the *same* thread, i.e., $\mathsf{thread} = \mathsf{thread}'$, then we have the same antecedent as for the sequential language. If the assertion and the assignment belong to *different* threads, interference freedom must be shown in any case except for the self-invocation of the start-method: The precondition of such a method invocation cannot interfere with the corresponding observation of the callee. To describe this setting, we define $\mathsf{self\_start}(q, \vec{y} := \vec{e})$ by $\mathsf{caller} = (\mathsf{this}, \mathsf{conf}', \mathsf{thread}')$ iff q is the precondition of a method invocation $e_0.\mathsf{start}(\vec{e})$ and the assignment is the callee observation at the beginning of the run-method, and by false otherwise.

For the example of Fig. 4, both p_2 and p_4, describing a thread different from the executing one, have to be invariant under the assignment. Also p_7 has to be invariant, if the assignment does not observe return. The assertion p_8 does not have to be invariant, where satisfaction of p_9 after execution is assured by local correctness. We don't have to show invariance of p_1, p_3, p_5, and p_6, since they describe objects different from the one in which the assignment is executed.

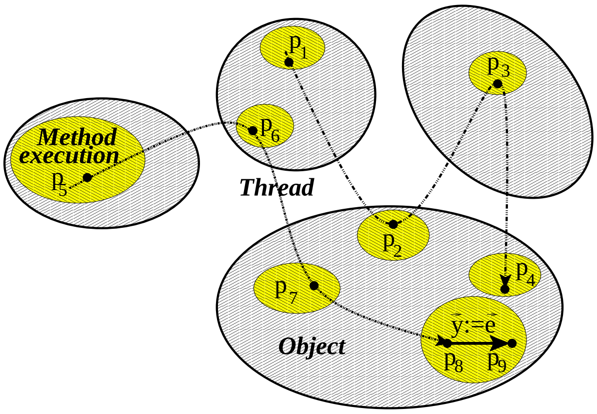


Fig. 4. Interference between threads

Definition 6 (Interference freedom). *A proof outline is* interference free, *if the conditions of Definition 3 hold with* $\mathsf{waits\_for\_ret}(q, \vec{y} := \vec{e})$ *replaced by*

$$\begin{aligned}\mathsf{interleavable}(q, \vec{y} := \vec{e}) \stackrel{def}{=} & \ \mathsf{thread} = \mathsf{thread}' \rightarrow \mathsf{waits\_for\_ret}(q, \vec{y} := \vec{e}) \wedge \\ & \ \mathsf{thread} \neq \mathsf{thread}' \rightarrow \neg \mathsf{self\_start}(q, \vec{y} := \vec{e}) \,.\end{aligned}$$

Example 14. Assume an assignment $\{p\}\ stm$ in an annotated method m of c, and an assertion q at a control point in the same method, which is not waiting for return, such that both p and q imply $\mathsf{thread} = \mathsf{this}$. I.e., the method is executed only by the thread of the object to which it belongs. Clearly, p and q cannot be simultaneously reached by the same thread. For invariance of q under the assignment stm, the antecedent of the interference freedom condition implies $p \wedge q' \wedge \mathsf{interleavable}(q, stm)$. From $p \wedge q'$ we conclude $\mathsf{thread} = \mathsf{thread}'$, and thus by the definition of $\mathsf{interleavable}(q, stm)$ the assertion q should be at a control point waiting for return, which is not the case, and thus the antecedent of the condition evaluates to false.

The cooperation test The cooperation test for object creation is not influenced by adding concurrency, but we have to extend the cooperation test for communication by defining additional conditions for thread creation. Invoking the start-method of an object whose thread is already started does not have communication effects. The same holds for returning from a run-method, which is already included in the conditions for the sequential language as for the termination of the only thread. Note that this condition applies now to all threads.

Definition 7 (Cooperation test: Communication). *A proof outline satisfies the* cooperation test for communication, *if the conditions of Definition 4 hold for the statements listed there with* $m \neq \mathsf{start}$, *and additionally in the following cases:*

1. CALL_{start}: *For all statements* $\{p_1\}\ e_0.\mathsf{start}(\vec{e})\ \ \{p_2\}^{!call}\langle \vec{y}_1 := \vec{e}_1\rangle^{!call}\{p_3\}$ *in class* c *with* e_0 *of type* c', comm *is given by* $E_0(z) = z' \wedge \neg z'.\mathsf{started}$, *where* $\{q_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{q_3\}\ stm; \mathsf{return}$ *is the body of the* run-*method of* c' *having formal parameters* $\vec{u}$, *and local variables* $\vec{v}$ *except the formal parameters. The callee class invariant is* $q_1 = I_{c'}$. *Furthermore,* f_{comm} *is* $\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v})$, f_{obs1} *is* $z.\vec{y}_1 := \vec{E}_1(z)$, *and* f_{obs2} *is* $z'.\vec{y}_2' := \vec{E}_2'(z')$.
2. $\text{CALL}^{skip}_{start}$: *For the above statements, the equations must additionally hold with the assertion* comm *given by* $E_0(z) = z' \wedge z'.\mathsf{started}$, $q_2 = q_3 = \mathsf{true}$, q_1 *and* f_{obs1} *as above, and* f_{comm} *and* f_{obs2} *are the empty statement.*

4 Reentrant monitors

In this section we extend the concurrent language with *monitor synchronization*. Again, we define syntax and semantics of the language Java_{synch} , before formalizing the proof system.

As a mechanism of concurrency control, methods can be declared as *synchronized.* Each object has a *lock* which can be owned by at most one thread. Synchronized methods of an object can be invoked only by a thread which owns the lock of that object. If the thread does not own the lock, it has to wait until the lock gets free. A thread owning the lock of an object can recursively invoke several synchronized methods of that object, which corresponds to the notion of reentrant monitors.

Besides mutual exclusion, using the lock-mechanism for synchronized methods, objects offer the methods wait, notify, and notifyAll as means to facilitate efficient thread coordination at the object boundary. A thread owning the lock of an object can block itself and free the lock by invoking wait on the given object. The blocked thread can be reactivated by another thread owning the lock via the object's notify method; the reactivated thread must re-apply for the lock before it may continue its execution. The method notifyAll, finally, generalizes notify in that it notifies all threads blocked on the object.

4.1 Syntax

Expressions and statements can be constructed as in the previous languages. The abstract syntax of the remaining constructs is summarized in Table 9.

$$
\begin{array}{rcl}
\mathit{modif} & ::= & \mathsf{nsync} \mid \mathsf{sync} \\
\mathit{meth} & ::= & \mathit{modif}\, m(u,\ldots,u)\{\ \mathit{stm}; \mathsf{return}\ \mathit{exp}_{ret}\} \\
\mathit{meth}_{\mathsf{run}} & ::= & \mathsf{nsync\ run}()\{\ \mathit{stm}; \mathsf{return}\ \} \\
\mathit{meth}_{\mathsf{wait}} & ::= & \mathsf{nsync\ wait}()\{\ ?\mathsf{signal}; \mathsf{return}_{getlock}\ \} \\
\mathit{meth}_{\mathsf{notify}} & ::= & \mathsf{nsync\ notify}()\{\ !\mathsf{signal}\ ; \mathsf{return}\ \} \\
\mathit{meth}_{\mathsf{notifyAll}} & ::= & \mathsf{nsync\ notifyAll}()\{\ !\mathsf{signal\_all}; \mathsf{return}\ \} \\
\mathit{meth}_{predef} & ::= & \mathit{meth}_{\mathsf{start}}\ \mathit{meth}_{\mathsf{wait}}\ \mathit{meth}_{\mathsf{notify}}\ \mathit{meth}_{\mathsf{notifyAll}} \\
\mathit{class} & ::= & \mathsf{class}\ c\{\mathit{meth}\ldots\mathit{meth}\ \mathit{meth}_{\mathsf{run}}\ \mathit{meth}_{predef}\} \\
\mathit{class}_{\mathsf{main}} & ::= & \mathit{class} \\
\mathit{prog} & ::= & \langle \mathit{class}\ldots\mathit{class}\ \mathit{class}_{\mathsf{main}}\rangle
\end{array}
$$

Table 9. $Java_{synch}$ abstract syntax

Methods get decorated by a modifier *modif* distinguishing between *non-synchronized* and *synchronized* methods.[10] In the sequel we also refer to statements in the body of a synchronized method as being synchronized. Furthermore, we consider the additional predefined methods wait, notify, and notifyAll, whose definitions use the auxiliary statements !signal, !signal_all, ?signal, and $\mathsf{return}_{getlock}$.[11]

[10] *Java* does not have the "non-synchronized" modifier: methods are non-synchronized by default.

[11] *Java*'s `Thread` class additionally support methods for suspending, resuming, and stopping a thread, but they are deprecated and thus not considered here.

4.2 Semantics

The operational semantics extends the semantics of $Java_{conc}$ by the rules of Table 10, where the CALL rule is replaced. For synchronized method calls, the

$$\frac{\begin{array}{c} m \notin \{\mathsf{start}, \mathsf{run}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\} \qquad modif\, m(\vec{u})\{\ body\ \} \in Meth_c \\ \beta = [\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in Val^c(\sigma) \qquad \tau' = \tau_{init}^{m,c}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \qquad (modif{=}\mathsf{sync}) \to \neg owns(T, \beta) \end{array}}{\begin{array}{l} \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, u := e_0.m(\vec{e}); stm)\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathsf{receive}\ u; stm) \circ (\beta, \tau', body)\}, \sigma\rangle \end{array}} \ \textsc{Call}$$

$$\frac{\begin{array}{c} m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\} \\ \beta = [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in Val^c(\sigma) \qquad owns(\xi \circ (\alpha, \tau, e.m(); stm), \beta) \end{array}}{\begin{array}{l} \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, e.m(); stm)\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathsf{receive}; stm) \circ (\beta, \tau_{init}^{m,c}, body_{m,c})\}, \sigma\rangle \end{array}} \ \textsc{Call}_{monitor}$$

$$\frac{\neg owns(T, \beta)}{\begin{array}{l} \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathsf{receive}; stm) \circ (\beta, \tau', \mathsf{return}_{getlock})\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, stm)\}, \sigma\rangle \end{array}} \ \textsc{Return}_{wait}$$

$$\frac{}{\begin{array}{l} \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, !\mathsf{signal}; stm)\} \,\dot\cup\, \{\xi' \circ (\alpha, \tau', ?\mathsf{signal}; stm')\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, stm)\} \,\dot\cup\, \{\xi' \circ (\alpha, \tau', stm')\}, \sigma\rangle \end{array}} \ \textsc{Signal}$$

$$\frac{wait(T, \alpha) = \emptyset}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, !\mathsf{signal}; stm)\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, stm)\}, \sigma\rangle} \ \textsc{Signal}_{skip}$$

$$\frac{T' = signal(T, \alpha)}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, !\mathsf{signal\_all}; stm)\}, \sigma\rangle \longrightarrow \langle T' \,\dot\cup\, \{\xi \circ (\alpha, \tau, stm)\}, \sigma\rangle} \ \textsc{SignalAll}$$

Table 10. $Java_{synch}$ Operational semantics

lock of the callee object has to be free or owned by the executing thread, as expressed by the predicate *owns*, defined below.

The remaining rules handle the semantics of the monitor methods wait, notify, and notifyAll. In all three cases the caller must own the lock of the callee object (rule $\textsc{Call}_{monitor}$). A thread can block itself on an object whose lock it owns by invoking the object's wait-method, thereby relinquishing the lock and placing itself into the object's wait set. Formally, the wait set $wait(T, \alpha)$ of an object is given as the set of all stacks in T with a top element of the form $(\alpha, \tau, ?\mathsf{signal}; stm)$. After having put itself on ice, the thread awaits notification by another thread which invokes the notify-method of the object. The !signal statement in the notify-method thus reactivates a non-deterministically chosen single thread waiting for notification on the given object (rule SIGNAL). Analogously to the wait set, the notified set $notified(T, \alpha)$ of α is the set of all stacks in T with top element of the form $(\alpha, \tau, \mathsf{return}_{getlock})$, i.e., threads which have been notified and are trying to get hold of the lock again. According to rule RETURN_{wait}, the receiver can continue after notification in executing $\mathsf{return}_{getlock}$ only if the lock is free. Note that the notifier does not hand over the lock to the one being notified but continues to own it. This behavior is known as *signal-and-continue* monitor discipline [And00]. If no threads are waiting on the object, the !signal of the notifier is without effect (rule SIGNAL_{skip}). The notifyAll-method generalizes notify in that all waiting threads are notified via the !signal_all-broadcast (rule SIGNALALL). The effect of this statement is given by defining $signal(T, \alpha)$ as $(T \setminus wait(T, \alpha)) \cup \{\xi \circ (\beta, \tau, stm) \mid \xi \circ (\beta, \tau, ?\mathsf{signal}; stm) \in wait(T, \alpha)\}$.

Using the wait and notified sets, we can now formalize the *owns* predicate: A thread ξ owns the lock of β iff ξ executes some synchronized method of β, but not its wait-method. Formally, $owns(T, \beta)$ is true iff there exists a thread $\xi \in T$ and a $(\beta, \tau, stm) \in \xi$ with stm synchronized and $\xi \notin wait(T, \beta) \cup notified(T, \beta)$. The definition is used analogously for single threads. An invariant of the semantics is that at most one thread can own the lock of an object at a time.

4.3 The proof system

The proof system has additionally to accommodate for synchronization, reentrant monitors, and thread coordination. First we define how to extend the augmentation of $Java_{conc}$, before we describe the proof method.

Proof outlines To capture mutual exclusion and the monitor discipline, the instance variable lock of type Thread $\times$ Int stores the identity of the thread who owns the lock, if any, together with the number of synchronized calls in its call chain. The initial lock value $free = (null, 0)$ indicates that the lock is free. The instance variables wait and notified of type list(Thread $\times$ Int) are the analogues of the *wait*- and *notified*-sets of the semantics and store the threads waiting at the monitor, respectively those having been notified. Besides the thread identity, the number of synchronized calls is stored. In other words, these variables remember the old lock-value prior to suspension which is restored when the thread becomes active again. All auxiliary variables are initialized as usual. For values *thread* of type Thread and *wait* of type list(Thread $\times$ Int), we will also write $thread \in wait$ instead of $(thread, n) \in wait$ for some n. If the order of the elements of a sequence is not relevant, we apply also set theoretical operations to them.

Syntactically, besides the augmentation of the previous section, the callee observation at the beginning and at the end of each synchronized method body executes lock := inc(lock) and lock := dec(lock), respectively. The semantics of incrementing the lock $[\![\mathsf{inc}(\mathsf{lock})]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}$ is $(\tau(\mathsf{thread}), n{+}1)$ for $\sigma_{inst}(\mathsf{lock}) = (\alpha, n)$. Decrementing $\mathsf{dec}(\mathsf{lock})$ is inverse: $[\![\mathsf{dec}(\mathsf{lock})]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}$ with $\sigma_{inst}(\mathsf{lock}) = (\alpha, n)$ is $(\alpha, n-1)$ if $n > 1$, and *free* otherwise.

Instead of the auxiliary statements of the semantics, notification is represented in the proof system by auxiliary assignments operating on the wait and notified variables. That means, the auxiliary ?signal, !signal, and !signal_all statements get replaced by auxiliary assignments[12] Entering the wait-method gets the observation $\mathsf{wait}, \mathsf{lock} := \mathsf{wait} \cup \{\mathsf{lock}\}, \mathsf{free}$; returning from the wait-method observes $\mathsf{lock}, \mathsf{notified} := \mathsf{get}(\mathsf{notified}, \mathsf{thread}), \mathsf{notified} \backslash \{\mathsf{get}(\mathsf{notified}, \mathsf{thread})\}$. For a thread $\alpha \in \mathit{Val}^{\mathsf{Thread}}$ and a list $\mathit{notified} \in \mathit{Val}^{\mathsf{list}(\mathsf{Thread} \times \mathsf{Int})}$, $\mathit{get}(\mathit{notified}, \alpha)$ retrieves the value (α, n) from the list. The semantics assures uniqueness of the association. The !signal statement of the notify-method is represented by the auxiliary assignment $\mathsf{wait}, \mathsf{notified} := \mathsf{notify}(\mathsf{wait}, \mathsf{notified})$, where the value $\mathit{notify}(\mathit{wait}, \mathit{notified})$ is the pair of the given sets with one element, chosen nondeterministically, moved from the wait into the notified set; if the wait set is empty, it is the identity function. Finally, the !signal_all statement of the notifyAll-method is represented by the auxiliary assignment $\mathsf{notified}, \mathsf{wait} := \mathsf{notified} \cup \mathsf{wait}, \emptyset$.

Verification conditions Initial and local correctness agree with those for Java_{conc}. In case of notification, local correctness covers also invariance for the notifying thread, as the effect of notification is captured by an auxiliary assignment.

The interference freedom test Synchronized methods of a single object can be executed concurrently only if one of the corresponding local configurations is waiting for return: If the executing threads are different, then one of the threads is in the wait or notified set of the object; otherwise, both executing local configurations are in the same call chain. Thus we assume that either not both the assignment and the assertion occur in a synchronized method, or the assertion is at a control point waiting for return.[13]

Definition 8 (Interference freedom). *A proof outline is* interference free, *if Definition 6 holds in all cases, such that either not both p and q occur in a synchronized method, or q is at a control point waiting for return.*

For notification, we require also invariance of the assertions for the notified thread. We do so, as notification is described by an auxiliary assignment executed by the notifier. That means, both the waiting and the notified status of the suspended thread are represented by a single control point in the wait-method. The two statuses can be distinguished by the values of the wait and notified

[12] In *Java*, the implementation of the monitor methods are syntactically not included in class definitions. Their augmentation and annotation can be specified by special comments.

[13] This condition is not necessary for a minimal proof system, but reduces the number of verification conditions.

variables. The invariance of the precondition of the return statement in the wait-method under the assignment in the notify-method represents the notification process, whereas invariance of that assertion over assignments changing the lock represents the synchronization mechanism. Information about the lock value will be imported from the cooperation test as this information depends on the global behavior.

Example 15. This example shows how the fact, that at most one thread can own the lock of an object, can be used to show mutual exclusion. We use the assertion $\mathsf{owns}(\mathsf{thread}, \mathsf{lock})$ for $\mathsf{thread} \neq \mathsf{null} \wedge \mathsf{thread}(\mathsf{lock}) = \mathsf{thread}$, where $thread(lock)$ is the first component of the lock value. Let furthermore $\mathsf{free\_for}(\mathsf{thread}, \mathsf{lock})$ be $\mathsf{thread} \neq \mathsf{null} \wedge (\mathsf{owns}(\mathsf{thread}, \mathsf{lock}) \vee \mathsf{lock} = \mathsf{free})$.

Let q, given by $\mathsf{owns}(\mathsf{thread}, \mathsf{lock})$, be an assertion at a control point and let $\{p\}^{?call}\langle stm\rangle^{?call}$ with $p \stackrel{def}{=} \mathsf{free\_for}(\mathsf{thread}, \mathsf{lock})$ be the callee observation at the beginning of a synchronized method in the same class. Note that the observation stm changes the lock value. The interference freedom condition $\models_{\mathcal{L}} \{p \wedge q' \wedge \mathsf{interleavable}(q, stm)\} stm \{q'\}$ assures invariance of q under the observation stm. The assertions p and q' imply $\mathsf{thread} = \mathsf{thread}'$. The points at p and q can be simultaneously reached by the same thread only if q describes a point waiting for return. This fact is mirrored by the definition of the interleavable predicate: If q is not at a control point waiting for return, then the antecedent of the condition evaluates to false. Otherwise, after the execution of the built-in augmentation $\mathsf{lock} := \mathsf{inc}(\mathsf{lock})$ in stm we have $\mathsf{owns}(\mathsf{thread}, \mathsf{lock})$, i.e., $\mathsf{owns}(\mathsf{thread}', \mathsf{lock})$, which was to be shown.

The cooperation test We extend the cooperation test for Java_{conc} with synchronization and the invocation of the monitor methods. In the previous languages, the assertion comm expressed, that the given statements indeed represent communicating partners. In the current language with monitor synchronization, communication is not always enabled. Thus the assertion comm has additionally to capture enabledness of the communication: In case of a synchronized method invocation, the lock of the callee object has to be free or owned by the caller. This is expressed by $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$, where thread is the caller thread, z' is the callee object, and where $\mathsf{thread}(z'.\mathsf{lock})$ is the first component of the lock value, i.e., the thread owning the lock of z'. For the invocation of the monitor methods we require that the executing thread is holding the lock. Returning from the wait-method assumes that the thread has been notified and that the callee's lock is free. Note that the global invariant is not affected by the object-internal monitor signaling mechanism, which is represented by auxiliary assignments.

Definition 9 (Cooperation test: Communication). *A proof outline satisfies the* cooperation test for communication*, if the conditions of Definition 7 hold for the statements listed there with the exception of the* CALL*-case, and additionally in the following cases:*

1. *Call: For all statements* $\{p_1\}\, u_{ret} := e_0.m(\vec{e})\ \{p_2\}^{!call} \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \{p_3\}^{wait}$ *(or such without receiving a value) in class c with e_0 of type c', where method $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ of c' is synchronized with body* $\{q_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{q_3\}\, stm; \mathsf{return}\ e_{ret}$*, formal parameters $\vec{u}$, and local variables $\vec{v}$ except the formal parameters. The callee class invariant is $q_1 = I_{c'}$. The assertion* comm *is given by* $E_0(z) = z' \wedge (z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread})$*. Furthermore, f_{comm} is* $\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v})$*, f_{obs1} is given by* $z.\vec{y}_1 := \vec{E}_1(z)$*, and f_{obs2} is* $z'.\vec{y}_2' := \vec{E}_2'(z')$*. If m is not synchronized,* $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ *in* comm *is dropped.*
2. *Call$_{monitor}$: For* $m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$*,* comm *is given by* $E_0(z) = z' \wedge \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$*.*
3. *Return$_{wait}$: For* $\{q_1\}\ \mathsf{return}_{getlock}\ \{q_2\}^{!ret} \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret} \{q_3\}$ *in a* wait*-method,* comm *is* $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge z'.\mathsf{lock} = \mathsf{free} \wedge \mathsf{thread}' \in z'.\mathsf{notified}$*.*

Example 16. Assume the invocation of a synchronized method m of a class c, where m of c has the body $\langle stm \rangle^{?call} \{\mathsf{thread}(\mathsf{lock}) = \mathsf{thread}\}\, stm'; \mathsf{return}$. Note that the built-in augmentation in stm sets the lock owner by the assignment $\mathsf{lock} := \mathsf{inc}(\mathsf{lock})$. Omitting irrelevant details again, the cooperation test requires $\models_{\mathcal{G}} \{\mathsf{true}\} z'.\mathsf{lock} := \mathsf{inc}(z'.\mathsf{lock}) \{\mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}'\}$, which holds by the definition of *inc*.

5 Exception handling

In this section we extend the previous language with *exception handling*. Again, we define syntax and semantics of the language Java$_{exc}$, before formalizing the proof system.

5.1 Syntax

We introduce additional statements for exception throwing and handling, as shown in Table 9. The abstract syntax of the remaining constructs is as for the previous language.

$$
\begin{array}{rl}
stm ::= & x := e \mid u := e \mid u := \mathsf{new}^c \\
\mid & u := e.m(e, \ldots, e) \mid e.m(e, \ldots, e) \\
\mid & \mathsf{throw}\ e \mid \mathsf{try}\ stm; \mathsf{catch}\ (c\, u)\ stm; \ldots \mathsf{catch}\ (c\, u)\ stm; \mathsf{finally}\ stm\ \mathsf{yrt} \\
\mid & \epsilon \mid stm; stm \mid \mathsf{if}\ e\ \mathsf{then}\ stm\ \mathsf{else}\ stm\ \mathsf{fi} \mid \mathsf{while}\ e\ \mathsf{do}\ stm\ \mathsf{od} \ldots
\end{array}
$$

Table 11. Java$_{exc}$ abstract syntax

$$\frac{\tau' = \tau[\mathsf{exc} \mapsto \tau(\mathsf{exc}) \circ \mathit{null}]}{\begin{array}{l}\langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{try}\ stm_0; \mathsf{catch}\ (c_1\ u_1)\ stm_1; \ldots; \mathsf{catch}\ (c_n\ u_n)\ stm_n; \mathsf{finally}\ stm_{n+1}\ \mathsf{yrt}; stm')\}, \sigma\rangle \longrightarrow \\ \langle T \dot\cup \{\xi \circ (\alpha, \tau', \quad stm_0; \mathsf{catch}\ (c_1\ u_1)\ stm_1; \ldots; \mathsf{catch}\ (c_n\ u_n)\ stm_n; \mathsf{finally}\ stm_{n+1}\ \mathsf{yrt}; stm')\}, \sigma\rangle\end{array}}\ \textsc{Try}$$

$$\frac{0 \leq n}{\begin{array}{l}\langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{catch}\ (c_1\ u_1)\ stm_1; \ldots; \mathsf{catch}\ (c_n\ u_n)\ stm_n; \mathsf{finally}\ stm_{n+1}\ \mathsf{yrt}; stm)\}, \sigma\rangle \longrightarrow \\ \langle T \dot\cup \{\xi \circ (\alpha, \tau, stm_{n+1}\ \mathsf{yrt}; stm)\}, \sigma\rangle\end{array}}\ \textsc{Finally}$$

$$\frac{\begin{array}{c}\tau(\mathsf{exc}) = \beta_0 \circ \ldots \circ \beta_k \circ \beta_{k+1} \qquad \tau' = \tau[\mathsf{exc} \mapsto \beta_0 \circ \ldots \circ \beta_k][\mathsf{top} \mapsto \beta_{k+1}] \\ \mathsf{if}\ \tau'(\mathsf{top}) = \mathit{null}\ \mathsf{then}\ stm' = stm\ \mathsf{else}\ stm' = \mathsf{throw\ top}; stm\ \mathsf{fi}\end{array}}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{yrt}; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau', stm')\}, \sigma\rangle}\ \textsc{Yrt}$$

Table 12. $Java_{exc}$ Operational semantics (1)

5.2 Semantics

Exceptions allow a special form of error handling: If something unexpected or unallowed happens, the executing thread may throw an exception, which is an object of an arbitrary[14] type. The empty reference cannot be thrown.[15] If an exception has been thrown by a thread, then the normal flow of control gets interrupted, and control tries to find the "nearest" exception handler handling exceptions of the given type, as explained below.

The operational semantics extends the semantics of $Java_{synch}$ by the rules of the Tables 12 and 13, covering exception handling. In the semantics of exception handling we add the type Object as the supertype of all classes. Note that no objects of type Object can be created, thus preserving monomorphism.

Throwing and catching exceptions are syntactically represented by throw statements and by try-catch-finally blocks. During the execution of a try-catch-finally block try stm_0; catch $(c_1\ u_1)\ stm_1; \ldots$; catch $(c_n\ u_n)\ stm_n$; finally stm_{n+1} yrt, the corresponding local configuration contains an "open" try-construct like e.g. stm'_0; catch $(c_1\ u_1)\ stm_1; \ldots$; catch $(c_n\ u_n)\ stm_n$; finally stm_{n+1} yrt (rule Try). We call such blocks also statements, even if they are no statements in a strong syntactical sense.[16] Statements in which no such open try blocks occur are called *try-closed.*

The semantics uses the local variable exc of types list Object with initial value ϵ, to store thrown but not yet caught exceptions. In nested try-catch-finally statements, each try-catch-finally statement has its own element in the sequence

[14] In *Java* only objects extending `Throwable` may be thrown.
[15] In *Java*, a `NullPointerException` is thrown in this case.
[16] Note that for example catch $(c_2\ u_2)\ stm_2$ is not a statement.

$$\frac{\begin{array}{c} stm \text{ is try-closed} \qquad stm' = \mathsf{catch}\,(c_1\,u_1)\,stm_1;\ldots;\mathsf{catch}\,(c_n\,u_n)\,stm_n;\mathsf{finally}\,stm_{n+1}\,\mathsf{yrt} \\ 1 \le i \le n \qquad [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in Val^{c_i} \qquad \forall 1 \le j < i.\,[\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \notin Val^{c_j} \\ \tau' = \tau[u_i \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \end{array}}{\begin{array}{l} \langle T \dot{\cup} \{\xi \circ (\alpha,\tau,\mathsf{throw}\,e;stm;stm';stm'')\},\sigma\rangle \longrightarrow \\ \langle T \dot{\cup} \{\xi \circ (\alpha,\tau',stm_i;\mathsf{finally}\,stm_{n+1}\,\mathsf{yrt};stm'')\},\sigma\rangle \end{array}}\ \textsc{Throw}_1$$

$$\frac{\begin{array}{c} stm \text{ is try-closed} \qquad stm' = \mathsf{catch}\,(c_1\,u_1)\,stm_1;\ldots;\mathsf{catch}\,(c_n\,u_n)\,stm_n;\mathsf{finally}\,stm_{n+1}\,\mathsf{yrt} \\ [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \neq null \qquad 0 \le n \qquad \forall 1 \le i \le n.\,[\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \notin Val^{c_i} \\ \tau(\mathsf{exc}) = \beta_0 \circ \ldots \circ \beta_k \circ \beta_{k+1} \qquad \tau' = \tau[\mathsf{exc} \mapsto \beta_0 \circ \ldots \circ \beta_k \circ [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \end{array}}{\begin{array}{l} \langle T \dot{\cup} \{\xi \circ (\alpha,\tau,\mathsf{throw}\,e;stm;stm';stm'')\},\sigma\rangle \longrightarrow \\ \langle T \dot{\cup} \{\xi \circ (\alpha,\tau',stm_{n+1}\,\mathsf{yrt};stm'')\},\sigma\rangle \end{array}}\ \textsc{Throw}_2$$

$$\frac{\begin{array}{c} stm \text{ is try-closed} \\ [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \neq null \qquad \tau(\mathsf{exc}) = \beta_0 \circ \ldots \circ \beta_k \circ \beta_{k+1} \qquad \tau' = \tau[\mathsf{exc} \mapsto \beta_0 \circ \ldots \circ \beta_k \circ [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \end{array}}{\langle T \dot{\cup} \{\xi \circ (\alpha,\tau,\mathsf{throw}\,e;stm\,\mathsf{yrt};stm')\},\sigma\rangle \longrightarrow \langle T \dot{\cup} \{\xi \circ (\alpha,\tau',\mathsf{yrt};stm')\},\sigma\rangle}\ \textsc{Throw}_3$$

$$\frac{stm' \text{ is try-closed} \qquad [\![e]\!]_{\mathcal{E}}^{\sigma(\beta),\tau'} \neq null \qquad \tau'' = \tau[\mathsf{top} \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma(\beta),\tau'}]}{\begin{array}{l} \langle T \dot{\cup} \{\xi \circ (\alpha,\tau,\mathsf{receive}\,u_{ret};stm) \circ (\beta,\tau',\mathsf{throw}\,e;stm')\},\sigma\rangle \longrightarrow \\ \langle T \dot{\cup} \{\xi \circ (\alpha,\tau'',\mathsf{throw}\,\mathsf{top};stm)\},\sigma\rangle \end{array}}\ \textsc{Throw}_4$$

$$\frac{stm \text{ is try-closed} \qquad [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \neq null}{\langle T \dot{\cup} \{(\alpha,\tau,\mathsf{throw}\,e;stm;\mathsf{return})\},\sigma\rangle \longrightarrow \langle T \dot{\cup} \{(\alpha,\tau,\mathsf{return})\},\sigma\rangle}\ \textsc{Throw}_5$$

Table 13. $Java_{exc}$ Operational semantics (2)

exc which is used to remember if there is an exception throw in that block which is not yet caught; a null-reference means the absence of such an exception. The additional variable top of type Object is used to store the value of an exception which should be rethrown.

Entering a try-catch-finally block appends a null-reference to the value of exc, expressing that there is no thrown but not yet caught exception in that block (cf. rule Try).

The execution of a try-catch-finally block consists of the execution of the try statement until an exception is thrown or the try statement terminates. If an exception is thrown, and if there is a corresponding catch-clause handling exceptions of the given type, then this catch-clause (cf. rule $\textsc{Throw}_1$) and the finally clause (cf. rule Finally with $n = 0$) get executed. Otherwise, if no exceptions have been thrown (cf. rules Finally) or if there is no corresponding catch clause (cf. rule $\textsc{Throw}_2$), then the finally clause gets executed. Also throwing an exception in a catch-clause (cf. rule $\textsc{Throw}_2$ with $n = 0$) causes the control to move to the finally block. Throwing an exception in the finally-clause overwrites exceptions thrown in the try- or catch-clauses (cf. rule $\textsc{Throw}_3$).

Exiting a try-catch-finally block removes the last element of exc and stores it in the variable top (cf. rule $\textsc{Yrt}$). If the value of top is different from the null reference, i.e., if there was a thrown but not caught exception in the block, then the exception gets rethrown.

Throwing an exception outside try-catch-finally blocks causes the control to return to the caller, and to rethrow the exception there (cf. rule $\textsc{Throw}_4$). For run-methods, throwing such an exception terminates the executing thread (cf. rule $\textsc{Throw}_5$).

If, due to a thrown exception, control returns to the caller, and if the callee local configuration is the only one in the stack which executes a synchronized method of the callee object, then its termination gives the lock free like normal termination. This happens after evaluating the corresponding finally clause within the method, if any. Note that returning from a method due to exception handling does not hand over the return value as specified in the return statement.

5.3 The proof system

The proof system has to accommodate additionally for exception handling. First we define how to extend the augmentation of $Java_{synch}$, before we describe the proof method.

Proof outlines We extend the local and the global assertion language with assertions of the form $\mathsf{hastype}(e, c)$ and $\mathsf{hastype}(E, c)$, respectively, which state that the value of e respectively E is of type c; we need this construct to be able to express which type of expression has been thrown. Remember that the programming language is monomorph, and thus the association is unique.

Furthermore, we extend the syntax of augmentation and annotation of the previous section to exception throwing and handling statements. Augmentation and annotation for exception throwing via the throw statement is of the form

$$\{p_0\}\ \mathsf{throw}\, u\ \{p_1\}^{throw} \langle \vec{y} := \vec{e} \rangle^{throw} \{p_2\}\ .$$

Exception throwing and its observation are executed in a single computation step, in this order. The assertion p_0 is the precondition of the throw statement. Note that the control point annotated by the postcondition p_2 is not reachable. The assertion p_1 describes the auxiliary point directly after exception throwing and before its observation $\vec{y} := \vec{e}$.

Furthermore, we extend the augmentation and annotation of method call statements, in order to logically capture the control flow if control returns to the caller due to an exception, which gets rethrown:

$$
\begin{array}{llll}
\{p_0\} & u := e_0.m(\vec{e}) & \{p_1\}^{!call} & \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \\
\{p_2\}^{wait} & & \{p_3\}^{?ret} & \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} \\
\{p_4\}^{exc} & & \{p_5\}^{rethrow} & \langle \vec{y}_{\mathsf{thr}} := \vec{e}_{\mathsf{thr}} \rangle^{rethrow} \\
\{p_6\}\ . & & &
\end{array}
$$

Again, after control returns but before the corresponding observation the assertion p_3 should hold. If control returns due to an exception, the assertion p_4 should hold after the observation. In this case the exception has to be rethrown; p_5 describes the state directly after rethrowing the exception in top prior to its observation $\vec{y}_{\mathsf{thr}} := \vec{e}_{\mathsf{thr}}$. Note that this observation does not have a postcondition, because the control point after the observation is not reachable. Note furthermore that only p_0, p_2, p_4, and p_6 annotate a control points. If control returns due to normal method termination, the assertion p_6 should hold after the observation $\vec{y}_4 := \vec{e}_4$.

The augmentation and annotation of try-catch-finally statements is of the form

$$
\begin{array}{llllll}
\{p_0\} & \mathsf{try} & \{p_1\}^{try} & \langle \vec{y}_{\mathsf{try}} := \vec{e}_{\mathsf{try}} \rangle^{try} & \{p_2\}\ stm_{\mathsf{try}}; & \{p_3\} \\
 & \mathsf{catch}(c_1\ u_1) & & & \{p_4\}\ stm_1; & \{p_5\} \\
 & \ldots & & & & \\
 & \mathsf{catch}(c_n\ u_n) & & & \{p'_4\}\ stm_n; & \{p'_5\} \\
 & \mathsf{finally} & & & \{p_6\}\ stm_{n+1} & \{p_7\} \\
 & \mathsf{yrt} & \{p_8\}^{yrt} & \langle \vec{y}_{\mathsf{yrt}} := \vec{e}_{\mathsf{yrt}} \rangle^{yrt} & & \\
\{p_9\}^{exc} & & \{p_{10}\}^{rethrow} & \langle \vec{y}_{\mathsf{thr}} := \vec{e}_{\mathsf{thr}} \rangle^{rethrow} & & \\
\{p_{11}\}\ . & & & & &
\end{array}
$$

The assertion p_0 is the precondition of the try-catch-finally block. The assertion p_1 should hold after entering the try-block and before the corresponding observation $\vec{y}_{\mathsf{try}} := \vec{e}_{\mathsf{try}}$, where the assertion p_2 describes the control point after observation, and p_3 is the postcondition of the whole try-block. The pre- and postconditions of the first and of the last catch blocks are p_4 and p_5 respectively p'_4 and p'_5. The finally block has the pre- and postconditions p_6 and p_7. After exiting the finally block, p_8 should hold prior to the observation $\vec{y}_{\mathsf{yrt}} := \vec{e}_{\mathsf{yrt}}$ of exiting. If there is an exception to be rethrown, the assertion p_9 is required to hold after the observation of yrt, p_{10} should hold after rethrowing and prior to its observation $\vec{y}_{\mathsf{thr}} := \vec{e}_{\mathsf{thr}}$. Again, this observation does not have a postcondition, because the control point after the observation is not reachable. Note that p_1, p_8, and p_{10} annotate auxiliary points. If there is no exception to be rethrown, the assertion p_{11} should hold after exiting the finally-block and executing the corresponding observation.

Remember that the local variable exc of type list Object with initial value ϵ stores the thrown but not yet caught exceptions in nested try-catch-finally blocks. The variable top stores the value of an exception to be rethrown. We use the assertion thrown as a shortcut for $\mathsf{tail}(\mathsf{exc}) \neq \mathsf{null}$, where the function $tail(v)$ gives the last element of the sequence v. We use also the function $head(v)$ which returns the sequence v without its last element[17]. Note that the variables exc and top are *local*. In the concurrent setting, all threads have their own exception mechanism, which are independent of each other.

The augmentation for the built-in auxiliary variable lock gets extended to capture the case when a thread terminates the execution of a synchronized method due to a thrown exception: We additionally observe each throw statement outside try-catch-finally blocks in a synchronized method by the assignment $\mathsf{lock} := \mathsf{dec}(\mathsf{lock})$.

Since the global invariant should describe object-external behavior, we required that instance variables occurring in the global invariant may be changed by observations of communication or object creation only. Since the execution of throw statements outside try-catch-finally blocks cause the control to move to the caller, i.e., its effect is also object-external, the observations of such throw statements may also change the values of instance variables referred to in the global invariant.

Verification conditions Initial correctness and interference freedom agree with those for $Java_{synch}$. Note that exception throwing and handling do not modify instance states. Invariance under their observations, which are multiple assignments, is already included in the interference freedom test conditions of the previous section.

Local correctness Additionally to the local correctness conditions of the previous section, we introduce new conditions to cover the control flow of exception handling.

Entering a try block pushes an empty reference on the exception stack (cf. rule TRY); thus the precondition of a try-catch-finally statement should imply the precondition of the try block after entering the block and executing the observation of the try keyword as stated in Condition (11). Furthermore, the precondition of the observation should hold directly after entering the block, prior to the observation, as formalized in Condition (10).

If no exceptions has been thrown in a try or in a catch block, then after termination of the block execution continues in the finally block (cf. rule FINALLY); the postcondition of each try and catch block should imply the precondition of the finally block, as required by Condition (12).

Exiting the finally block (cf. rule YRT) is covered by the Conditions (13)-(15). Condition (13) assures that p_{yrt} holds after exiting the finally block but before its observation. Remember that in case of a thrown but not yet caught exception the exception is stored in the variable top, and becomes rethrown after the block; in this case the assertion p_{exc} is required to hold after the observation of yrt and prior to rethrowing, as stated in Condition (15). If no exceptions must be rethrown, Condition (14) assures that the assertion p' is satisfied after the termination of the try-catch-finally block.

[17] These functions are applied to non-empty sequences only.

If an exception has been thrown in a try block (cf. rules $\textsc{Throw}_1$ and $\textsc{Throw}_2$), then the precondition of the throw statement must imply the precondition of the corresponding catch block, if any, after throwing and its observation, and the precondition of the finally block otherwise; these cases are covered by the Conditions (17) and (19). Satisfaction of the preconditions of the corresponding observations is covered by the Conditions (16) and (18). The conditions for exception throwing in a catch block, in a finally block, or outside try-catch-finally blocks in run methods are modifications of the above conditions.

Remember that if an exception is thrown but not yet caught, the execution will not continue after the try-catch-finally block, but move to the next outer try-catch-finally block or to the caller configuration. The latter (cf. rule $\textsc{Throw}_4$) is covered by the conditions of the cooperation test for exception handling.

Definition 10 (Local correctness: Exception handling). *A proof outline is* locally correct *under exception handling, if for all statements stm of the form*

$$\begin{array}{llllll}
\{p\} & \mathsf{try} & \{p_{\mathsf{try}}\}^{try} & \langle \vec{y}_{\mathsf{try}} := \vec{e}_{\mathsf{try}}\rangle^{try} & \{p_0\} & stm_{\mathsf{try}};\ \{p'_0\} \\
 & \mathsf{catch}(c_1\, u_1) & & & \{p_1\} & stm_1;\ \{p'_1\} \\
 & \ldots & & & & \\
 & \mathsf{catch}(c_n\, u_n) & & & \{p_n\} & stm_n;\ \{p'_n\} \\
 & \mathsf{finally} & & & \{p_{\mathsf{fin}}\} & stm_{\mathsf{fin}}\ \ \{p'_{\mathsf{fin}}\} \\
 & \mathsf{yrt} & \{p_{\mathsf{yrt}}\}^{yrt} & \langle \vec{y}_{\mathsf{yrt}} := \vec{e}_{\mathsf{yrt}}\rangle^{yrt} & & \\
\{p_{\mathsf{exc}}\}^{exc} & & \{p_{\mathsf{thr}}\}^{rethrow} & \langle \vec{y}_{\mathsf{thr}} := \vec{e}_{\mathsf{thr}}\rangle^{rethrow} & & \\
\{p'\}\ , & & & & &
\end{array}$$

and for all $0 \leq i \leq n$*,*

$$\models_{\mathcal{L}} \{p\}\ \mathsf{exc} := \mathsf{exc} \circ \mathsf{null}\ \{p_{\mathsf{try}}\}\,, \tag{10}$$
$$\models_{\mathcal{L}} \{p\}\ \mathsf{exc} := \mathsf{exc} \circ \mathsf{null};\ \vec{y}_{\mathsf{try}} := \vec{e}_{\mathsf{try}}\ \{p_0\}\,, \tag{11}$$
$$\models_{\mathcal{L}} p'_i \rightarrow p_{\mathsf{fin}}\,, \tag{12}$$
$$\models_{\mathcal{L}} \{p'_{\mathsf{fin}}\}\ \mathsf{exc}, \mathsf{top} := \mathsf{head}(\mathsf{exc}), \mathsf{tail}(\mathsf{exc})\ \{p_{\mathsf{yrt}}\}\,, \tag{13}$$
$$\models_{\mathcal{L}} \{p_{\mathsf{fin}'} \wedge \mathsf{tail}(\mathsf{exc}) = \mathsf{null}\}\ \mathsf{exc}, \mathsf{top} := \mathsf{head}(\mathsf{exc}), \mathsf{tail}(\mathsf{exc});\ \vec{y}_{\mathsf{yrt}} := \vec{e}_{\mathsf{yrt}}\ \{p'\}\,, \tag{14}$$
$$\models_{\mathcal{L}} \{p'_{\mathsf{fin}} \wedge \mathsf{tail}(\mathsf{exc}) \neq \mathsf{null}\}\ \mathsf{exc}, \mathsf{top} := \mathsf{head}(\mathsf{exc}), \mathsf{tail}(\mathsf{exc});\ \vec{y}_{\mathsf{yrt}} := \vec{e}_{\mathsf{yrt}}\ \{p_{\mathsf{exc}}\}\,, \tag{15}$$

and for all statements $\{q_0\}$ throw e $\{q_1\}^{throw}\langle\vec{y} := \vec{e}\rangle^{throw}$ *in* stm_{try} *which do not occur in an inner try-catch-finally block inside* stm_{try}*, and for all* $1 \leq i \leq n$*,*

$$\models_{\mathcal{L}} \{q_0 \wedge e \neq \mathsf{null} \wedge \mathsf{hastype}(e, c_i) \wedge \forall 1 \leq j < i. \neg\, \mathsf{hastype}(e, c_j)\} \quad u_i := e \quad \{q_1\}\,, \tag{16}$$

$$\models_{\mathcal{L}} \{q_0 \wedge e \neq \mathsf{null} \wedge \mathsf{hastype}(e, c_i) \wedge \forall 1 \leq j < i. \neg\, \mathsf{hastype}(e, c_j)\} \quad u_i := e; \quad \vec{y} := \vec{e} \quad \{p_i\}\,, \tag{17}$$

$$\models_{\mathcal{L}} \{q_0 \wedge e \neq \mathsf{null} \wedge \forall 1 \leq j \leq n. \neg\, \mathsf{hastype}(e, c_j)\} \quad \mathsf{exc} := \mathsf{head}(\mathsf{exc}) \circ e \quad \{q_1\}\,, \tag{18}$$

$$\models_{\mathcal{L}} \{q_0 \wedge e \neq \mathsf{null} \wedge \forall 1 \leq j \leq n. \neg\, \mathsf{hastype}(e, c_j)\} \quad \mathsf{exc} := \mathsf{head}(\mathsf{exc}) \circ e; \quad \vec{y} := \vec{e} \quad \{p_{\mathsf{fin}}\}\,. \tag{19}$$

For statements $\{q_0\}$ throw e $\{q_1\}^{throw}\langle\vec{y} := \vec{e}\rangle^{throw}$ *in catch blocks, (18) and (19) are required to hold without the antecedent* $\forall 1 \leq j \leq n. \neg\, \mathsf{hastype}(e, c_j)$*. For* throw *statements in finally blocks, (18) and (19) should hold without the above antecedent and with* p_{fin} *replaced by* p'_{fin}*. The above conditions (16)-(19) should hold also for statements of the form* $\{q_0\}^{exc}\{q_1\}^{rethrow}\langle\vec{y} := \vec{e}\rangle^{rethrow}$*, where the expression* e *in the conditions is replaced by* top*. Finally, for statements of the form* $\{q_0\}$ throw e $\{q_1\}^{throw}\langle\vec{y} := \vec{e}\rangle^{throw}$ *outside try-catch-finally blocks in a* run*-method with body* stm'; return*, (18) and (19) should hold without the above antecedent, with* p_{fin} *replaced by* $pre(\mathsf{return})$*, and without the update of* exc*. The above conditions must hold also for all statements* $\{q_0\}\{q_1\}^{rethrow}\langle\vec{y} := \vec{e}\rangle^{rethrow}$*, where the expression* e *in the conditions is replaced by* top*.*

The cooperation test To cover exception handling, we extend the cooperation test conditions for $Java_{synch}$ with additional conditions, collected in the *cooperation test for exception handling.* The cooperation test for exception handling covers exception throwing if it is not in the scope of any try-catch-finally block, i.e., if it causes the control to return to the caller configuration.

Assume a method call and a throw statement outside any try-catch-finally block in the invoked method:

caller: $u_{ret} := e_0.m(\vec{e}) \ldots \{p_1\}^{wait}$		$\{p_2\}^{?ret}$ $\langle\vec{y}_4 := \vec{e}_4\rangle^{?ret}$	$\{p_3\}^{exc} \ldots$
callee:	$\ldots \{q_1\}$	throw e $\{q_2\}^{throw}$ $\langle\vec{y}_3 := \vec{e}_3\rangle^{throw}$	$\ldots$

We assume that the global invariant, the precondition q_1 of the throw statement, and the assertion p_1 of the caller at the control point waiting for return hold prior to exception throwing. Exception throwing communicates the identity of the thrown exception. Directly after exception throwing, the preconditions p_2 and q_2 of the corresponding observations must hold, as required by Condition

(20) of the cooperation test below. After the throw statement, its observation, and the observation of the caller have been executed, the global invariant and the postcondition p_3 of the caller observation is required to hold, as formalized in Condition (21). Note that the control point after the callee observation is not reachable, thus the assertion at this point is not required to hold.

Let the fresh logical variables z and z' denote the caller respectively the callee object. Since these objects are in general different, the cooperation test is formulated in the global language. Local assertions are expressed in the global language using the lifting substitution. For example, the assertion p_1 of the caller is expressed on the global level by $P_1(z) = p_1[z/\mathsf{this}]$. To distinguish local variables of caller and callee, we rename those of the callee; the result we denote by primed variables, expressions, and assertions. For example, to reason about q_1 in the cooperation test we rename all local variables in q_1 resulting in q_1', where $Q_1'(z') = q_1'[z'/\mathsf{this}]$ is q_1' expressed in the global language.

That the identity of the thrown exception is stored in the local variable top of the caller is represented by the assignment $\mathsf{top} := E'(z')$. The callee and the caller observations are represented by the assignments $z'.\vec{y}_3' := \vec{E}_3'(z')$ and $z.\vec{y}_4 := \vec{E}_4(z)$, respectively. Note that if the invoked method is synchronized, than the observation $z'.\vec{y}_3' := \vec{E}_3'(z')$ decrements the value of the lock of z' by the built-in augmentation.

We use the assertion comm to express that the local configurations described by p_1 and q_1 are indeed communication partners: By $E_0(z) = z'$ we require that the value of z' is indeed the callee object of the invocation $e_0.m(\vec{e})$. Remember that method call statements must not contain instance variables, and that formal parameters must not be assigned to. That means, the values of e_0, and the values of the formal and actual parameters do not change during method evaluation. The assertion $\vec{u}' = \vec{E}(z)$ states that the values of the formal and of the actual parameters agree, which implies that the primed built-in auxiliary formal parameter caller' of the callee stores $(z, \mathsf{conf}, \mathsf{thread})$ identifying the caller. I.e., the assertion $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$ assures that the local configurations are in caller-callee relationship. Furthermore, $E'(z') \neq \mathsf{null}$ expresses that the exception to be thrown is not the null reference, i.e., that exception throwing is enabled.

Definition 11 (Cooperation test: Exception handling). *A proof outline satisfies the* cooperation test for exception handling, *if for all statements* $u_{ret} := e_0.m(\vec{e})\ \langle stm\rangle^{!call}\{p_1\}^{wait}\{p_2\}^{?ret}\langle\vec{y}_4 := \vec{e}_4\rangle^{?ret}\{p_3\}^{exc}$ *(or such without receiving a value) occurring in class c with $m \neq$* start *and e_0 of type c', and for all* $\{q_1\}\ \mathsf{throw}\ e\,\{q_2\}^{throw}\langle\vec{y}_3 := \vec{e}_3\rangle^{throw}$ *in $m(\vec{u})$ of c' which are not inside any* *try-catch-finally statement,*

$$\models_{\mathcal{G}} \qquad \{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm}\} \qquad (20)$$
$$\mathsf{top} := E'(z')$$
$$\{P_2(z) \wedge Q_2'(z')\} \quad \textit{and}$$
$$\models_{\mathcal{G}} \qquad \{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm}\} \qquad (21)$$
$$\mathsf{top} := E'(z'); \quad z'.\vec{y}_3' := \vec{E}_3'(z'); \quad z.\vec{y}_4 := \vec{E}_4(z)$$
$$\{GI \wedge P_3(z)\}$$

must hold with distinct fresh logical variables $z \in LVar^c$ and $z' \in LVar^{c'}$, and with comm *given by $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge E'(z') \neq \mathsf{null} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}$.*

Furthermore, the same conditions must hold also for statements of the form $\{q_1\}^{exc}\{q_2\}^{rethrow}\langle\vec{y}_3 := \vec{e}_3\rangle^{rethrow}$ under the same requirements, where e in the conditions is replaced by top.

Example 17. In the proof outline below, the main class `Inc` offers a method `inc`, which increases the value of the instance variable x, if its value is not 100, and throws an exception of type E, otherwise. Thus the first 100 invocations of `inc` will increase x, and each further invocation throws an exception.

The `run` method of the class calls `inc` in an infinite loop. Control can exit this loop only if an exception has been thrown. The proof outline satisfies the conditions of the proof system, and thus assures that, if the `run` method terminates, then x has the value 100. Unspecified assertions are by definition true. The built-in augmentation is not listed in the code. We explicitly define the instance and local variables in *Java*-style.

```
class Inc{
     int x;
     nsync run(){
          try while true do
                 inc() {x = 100 ∧ hastype(exc, E) }^exc
              od; {false}
          catch (E u); {x = 100}
          finally {x = 100} yrt; {x = 100}^exc {x = 100}
          return }
     sync inc(){
          E v;
          if x=100 then {x = 100}
            v := new E(); {x = 100 ∧ hastype(v, E) }
            throw v {false}
          fi; {x ≠ 100}
          x := x+1;
          return }
}

class E{...}
```

We only deal with the conditions for exception throwing and handling. The conditions for assertions which are by definition true are trivial. For the only try-catch-finally block, Condition (12) yields

$$\models_{\mathcal{L}} \mathsf{false} \rightarrow (x = 100)$$

for $i = 0$ and

$$\models_{\mathcal{L}} (x = 100) \rightarrow (x = 100)$$

for $i = 1$. Condition (14) for exiting the block states

$$\models_{\mathcal{L}} \{x = 100\}\ \mathsf{exc}, \mathsf{top} := \mathsf{head}(\mathsf{exc}), \mathsf{tail}(\mathsf{exc})\ \{x = 100\}\,.$$

For re-throwing after the method call in the run method, the local correctness Condition (17) requires

$$\models_{\mathcal{L}} \{x = 100 \wedge \mathsf{hastype}(\mathsf{top}, E)\}\ u := \mathsf{top}\ \{x = 100\}\,.$$

The antecedent of Condition (19) leads to a type contradiction. Rethrowing an exception after the try-catch-finally block terminates the given thread. The local correctness Condition 19 requires

$$(x = 100 \wedge \mathsf{top} \neq \mathsf{null}) \rightarrow (x = 100)$$

for this case.

The throw statement in the `inc` method is outside any try-catch-finally blocks, thus we have to apply the cooperation test to show inductivity. Condition 21 assures validity of the postcondition of the caller by the requirement

$$\begin{array}{l}\models_{\mathcal{G}} \{(z'.x = 100 \wedge \mathsf{hastype}(v', E)) \wedge z = z' \wedge v' \neq \mathsf{null} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\} \\ \qquad \mathsf{top} := v';\ z'.\mathsf{lock} := \mathsf{dec}(z'.\mathsf{lock}) \quad \{z.x = 100 \wedge \mathsf{hastype}(\mathsf{top}, E)\}\,.\end{array}$$

6 Weakest precondition calculus

The verification conditions of the previous sections were formulated as standard Hoare-triples. In this section we define their formal semantics, given by means of a weakest precondition calculus. To do so, first we introduce substitutions in Section 6.1, before re-formulating the verification conditions for $Java_{exc}$ in Section 6.2 to logical implications, using the substitutions. The proofs of the lemmas in this section can be found in Appendix A.

6.1 Substitution operations

The verification conditions defined in the next section involve three substitution operations: the local, the global, and the lifting substitution. The lifting substitution is already defined in Section 2.3. The local substitution will be used to express the effect of assignments in local assertions. The global substitution is used similarly for global assertions.

The *local substitution* $p[\vec{e}/\vec{y}]$ is the standard capture-avoiding substitution, replacing in the local assertion p all occurrences of the given distinct variables $\vec{y}$ by the local expressions $\vec{e}$. We apply the substitution also to local expressions. The following lemma expresses the standard property of the above substitution, relating it to state-update. The relation between substitution and update formulated in the lemma asserts that $p[\vec{e}/\vec{y}]$ is the *weakest precondition* of p wrt. to the assignment $\vec{y} := \vec{e}$. The lemma is formulated for assertions, but the same property holds for expressions.

Lemma 3 (Local substitution). *For arbitrary logical environments ω and instance local states (σ_{inst}, τ) we have*

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p[\vec{e}/\vec{y}] \quad \textit{iff} \quad \omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}], \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}] \models_{\mathcal{L}} p\,.$$

The effect of assignments is expressed on the global level by the *global substitution* $P[\vec{E}/z.\vec{x}]$, which replaces in the global assertion P the instance variables $\vec{x}$ of the object referred to by z by the global expressions $\vec{E}$. To accommodate properly for the effect of assignments, though, we must not only syntactically replace the occurrences $z.x_i$ of the instance variables, but also all their *aliases* $E'.x_i$, when z and the result of the substitution applied to E' refer to the same object. As the aliasing condition cannot be checked syntactically, we define the main case of the substitution by a conditional expression [AdB93]:

$$(E'.x_i)[\vec{E}/z.\vec{x}] = \ (\mathsf{if}\ \ E'[\vec{E}/z.\vec{x}] = z\ \ \mathsf{then}\ \ E_i\ \ \mathsf{else}\ (E'[\vec{E}/z.\vec{x}]).x_i\ \ \mathsf{fi})\ .$$

The substitution is extended to global assertions homomorphically. We will also use the substitution $P[\vec{E}/z.\vec{y}]$ for arbitrary variable sequences $\vec{y}$ possibly containing logical variables, whose semantics is defined by the simultaneous substitutions $[\vec{E}_x/z.\vec{x}]$ and $[\vec{E}_u/\vec{u}]$, where $\vec{x}$ and $\vec{u}$ are the sequences of the instance and logical variables[18] of $\vec{y}$, and $\vec{E}_x$ and $\vec{E}_u$ the corresponding subsequences of $\vec{E}$ and $[\vec{E}_u/\vec{u}]$ is the usual capture-avoiding substitution like in the local substitution; if only logical variables are substituted, we simply write $P[\vec{E}/\vec{u}]$. That the substitution accurately catches the semantical update, and thus represents the weakest precondition relation, is expressed by the following lemma:

Lemma 4 (Global substitution). *For arbitrary global states σ and logical environments ω referring only to values existing in σ we have*

$$\omega, \sigma \models_{\mathcal{G}} P[\vec{E}/z.\vec{y}] \quad \textit{iff} \quad \omega', \sigma' \models_{\mathcal{G}} P\ ,$$

where $\omega' = \omega[\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}]$ *and* $\sigma' = \sigma[[\![z]\!]_{\mathcal{G}}^{\omega,\sigma}.\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}]$.

6.2 Verification conditions

In the local verification conditions, the effect of an assignment $\vec{y} := \vec{e}$ is expressed by substituting $\vec{e}$ for $\vec{y}$ in the assertions. In the global conditions of the cooperation test, the effect of communication, changing local states only, is expressed by simultaneously substituting the variables, which will store the result, by the communicated values. I.e., for the case of method call, the formal parameters get replaced by the actual ones expressed in the global language. The effect of the caller observation $\langle \vec{y} := \vec{e} \rangle^{!call}$ to a global assertion P is expressed by the substitution $P[\vec{E}(z)/z.\vec{y}]$, where z represents the caller. The effect of the callee-observation is handled similarly. Note the order: first communication takes place, followed by the sender, and then the receiver observation. To describe the

[18] Local variables are viewed as logical ones in the global assertion language.

common effect, we first have to substitute for the receiver, then for the sender observation, and finally for communication. For method call, we additionally have to substitute for the initialization of the local variables.

For readability, in the following definitions we will use the notation $p \circ f$ with $f = [\vec{e}/\vec{y}]$ for the substitution $p[\vec{e}/\vec{y}]$; we use a similar notation for global assertions. Note that the substitution binds stronger than logical operators.

Definition 12 (Initial correctness). *A proof outline is* initially correct, *if*

$$\models_{\mathcal{G}} \mathsf{InitState}(z) \wedge (\forall z'.\ z' = \mathsf{null} \vee z = z') \rightarrow \\ P_2(z) \circ f_{init} \wedge (GI \wedge P_3(z) \wedge I_c(z)) \circ f_{obs} \circ f_{init}\ , \tag{22}$$

where c is the main class, $\{p_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{p_3\}\, stm; \mathsf{return}$ *is the body and* $\vec{v}$ *the local variables of the* run*-method of c,* $z \in LVar^c$*, and* $z' \in LVar^{\mathsf{Object}}$*. Furthermore,*

$$f_{init} = [z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]\ ,\ \textit{and} \\ f_{obs} = [\vec{E}_2(z)/z.\vec{y}_2]\,.$$

Definition 13 (Local correctness: Assignment). *A proof outline is* locally correct, *if for all multiple assignments* $\{p_1\}\ \vec{y} := \vec{e}\ \{p_2\}$ *in class c, being an unobserved assignment, an alone-standing observation, or an observed assignment,*

$$\models_{\mathcal{L}} p_1 \rightarrow p_2 \circ f_{ass}\ , \tag{23}$$

with $f_{ass} = [\vec{e}/\vec{y}]$.

Definition 14 (Local correctness: Exception handling). *A proof outline is* locally correct *under exception handling, if for all statements stm of the form*

$$\begin{array}{llllll}
\{p\} & \mathsf{try} & \{p_{\mathsf{try}}\}^{try} & \langle \vec{y}_{\mathsf{try}} := \vec{e}_{\mathsf{try}}\rangle^{try} & \{p_0\} & stm_{\mathsf{try}};\ \{p'_0\} \\
 & \mathsf{catch}(c_1\, u_1) & & & \{p_1\} & stm_1;\ \{p'_1\} \\
 & \ldots & & & & \\
 & \mathsf{catch}(c_n\, u_n) & & & \{p_n\} & stm_n;\ \{p'_n\} \\
 & \mathsf{finally} & & & \{p_{\mathsf{fin}}\} & stm_{\mathsf{fin}}\ \{p'_{\mathsf{fin}}\} \\
 & \mathsf{yrt} & \{p_{\mathsf{yrt}}\}^{yrt} & \langle \vec{y}_{\mathsf{yrt}} := \vec{e}_{\mathsf{yrt}}\rangle^{yrt} & & \\
\{p_{\mathsf{exc}}\}^{exc} & & \{p_{\mathsf{thr}}\}^{rethrow} & \langle \vec{y}_{\mathsf{throw}} := \vec{e}_{\mathsf{throw}}\rangle^{rethrow} & & \\
\{p'\}\ , & & & & &
\end{array}$$

and for all $0 \leq i \leq n$,

$$\models_{\mathcal{L}} p \rightarrow p_{\mathsf{try}}[\mathsf{exc} \circ \mathsf{null}/\mathsf{exc}] \wedge p_0[\vec{e}_{\mathsf{try}}/\vec{y}_{\mathsf{try}}][\mathsf{exc} \circ \mathsf{null}/\mathsf{exc}]\,, \tag{24}$$

$$\models_{\mathcal{L}} p'_i \rightarrow p_{\mathsf{fin}}\,, \tag{25}$$

$$\models_{\mathcal{L}} p'_{\mathsf{fin}} \rightarrow p_{\mathsf{yrt}}[\mathsf{head}(\mathsf{exc}), \mathsf{tail}(\mathsf{exc})/\mathsf{exc}, \mathsf{top}]\,, \tag{26}$$

$$\models_{\mathcal{L}} (p'_{\mathsf{fin}} \wedge \mathsf{tail}(\mathsf{exc}) = \mathsf{null}) \rightarrow p'[\vec{e}_{\mathsf{yrt}}/\vec{y}_{\mathsf{yrt}}][\mathsf{head}(\mathsf{exc}), \mathsf{tail}(\mathsf{exc})/\mathsf{exc}, \mathsf{top}]\,, \tag{27}$$

$$\models_{\mathcal{L}} (p'_{\mathsf{fin}} \wedge \mathsf{tail}(\mathsf{exc}) \neq \mathsf{null}) \rightarrow p_{\mathsf{exc}}[\vec{e}_{\mathsf{yrt}}/\vec{y}_{\mathsf{yrt}}][\mathsf{head}(\mathsf{exc}), \mathsf{tail}(\mathsf{exc})/\mathsf{exc}, \mathsf{top}]\,, \tag{28}$$

and for all statements $\{q_0\}$ $\mathsf{throw}\, e$ $\{q_1\}^{throw}\langle\vec{y} := \vec{e}\rangle^{throw}$ *in* stm_{try} *which does not occur in an inner try-catch-finally block inside* stm_{try}*, and for all* $1 \le i \le n$*,*

$$\models_{\mathcal{L}} (q_0 \wedge e \neq \mathsf{null} \wedge \mathsf{hastype}(e, c_i) \wedge \forall 1 \le j < i. \neg\, \mathsf{hastype}(e, c_j)) \rightarrow \quad (29)$$
$$q_1[e/u_i] \wedge p_i[\vec{e}/\vec{y}][e/u_i]\,,$$
$$\models_{\mathcal{L}} (q_0 \wedge e \neq \mathsf{null} \wedge \forall 1 \le j \le n. \neg\, \mathsf{hastype}(e, c_j)) \rightarrow \quad (30)$$
$$q_1[\mathsf{head(exc)} \circ e/\mathsf{exc}] \wedge p_{\mathsf{fin}}[\vec{e}/\vec{y}][\mathsf{head(exc)} \circ e/\mathsf{exc}]\,.$$

For statements $\{q_0\}$ $\mathsf{throw}\, e$ $\{q_1\}^{throw}\langle\vec{y} := \vec{e}\rangle^{throw}$ *in catch blocks, (30) is required to hold without the antecedent* $\forall 1 \le j \le n. \neg\, \mathsf{hastype}(e, c_j)$*. For* throw *statements in finally blocks, (30) should hold without the above antecedent and with* p_{fin} *replaced by* p'_{fin}*. The above conditions (29) and (30) should hold also for statements of the form* $\{q_0\}\{q_1\}^{rethrow}\langle\vec{y} := \vec{e}\rangle^{rethrow}$*, where the expression* e *in the conditions is replaced by* top*. Finally, for statements of the form* $\{q_0\}$ $\mathsf{throw}\, e$ $\{q_1\}^{throw}\langle\vec{y} := \vec{e}\rangle^{throw}$ *outside try-catch-finally blocks in a* run*-method with body* stm'; return*, Condition (30) should hold without the above antecedent, without the update of* exc*, and with the assertion* p_{fin} *replaced by* $pre(\mathsf{return})$*. For statements* $\{q_0\}\{q_1\}^{rethrow}\langle\vec{y} := \vec{e}\rangle^{rethrow}$ *the same conditions must hold where we additionally replace* e *by* top*.*

Definition 15 (Interference freedom). *A proof outline is* interference free*, if for all classes* c*, and for all multiple assignments* $\vec{y} := \vec{e}$ *with precondition* p *in* c*,*

$$\models_{\mathcal{L}} p \wedge I_c \rightarrow I_c \circ f_{ass}\ , \quad (31)$$

with $f_{ass} = [\vec{e}/\vec{y}]$*. Furthermore, for all assertions* q *at control points in* c*, such that either not both* p *and* q *occur in a synchronized method, or* q *is at a control point waiting for return,*

$$\models_{\mathcal{L}} p \wedge q' \wedge \mathsf{interleavable}(q, \vec{y} := \vec{e}) \rightarrow q' \circ f_{ass}\ . \quad (32)$$

Definition 16 (Cooperation test: Communication). *A proof outline satisfies the* cooperation test for communication*, if*

$$\models_{\mathcal{G}} GI \wedge P_1(z) \wedge Q'_1(z') \wedge \mathsf{comm} \wedge z{\neq}\mathsf{null} \wedge z'{\neq}\mathsf{null} \rightarrow$$
$$(P_2(z) \wedge Q'_2(z')) \circ f_{comm} \wedge$$
$$(GI \wedge P_3(z) \wedge Q'_3(z')) \circ f_{obs2} \circ f_{obs1} \circ f_{comm} \quad (33)$$

holds for distinct fresh logical variables $z \in LVar^c$ *and* $z' \in LVar^{c'}$*, in the following cases:*

1. (a) Call*: For all calls* $\{p_1\}\, u_{ret} := e_0.m(\vec{e})\ \{p_2\}^{!call}\langle\vec{y}_1 := \vec{e}_1\rangle^{!call}\{p_3\}^{wait}$ *(or such without receiving a value) in class* c *with* e_0 *of type* c'*, where method* $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ *of* c' *is synchronized with body* $\{q_2\}^{?call}\langle\vec{y}_2 := \vec{e}_2\rangle^{?call}\{q_3\}\ stm;$ return e_{ret}*, formal parameters* $\vec{u}$*, and local variables* $\vec{v}$ *except the formal parameters. The callee class invariant is* $q_1 = I_{c'}$*. The assertion* comm *is given by* $E_0(z) = z' \wedge (z'.\mathsf{lock} = \mathsf{free} \vee$

$\mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread})$. *Furthermore,* $f_{comm} = [\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}']$, $f_{obs1} = [\vec{E}_1(z)/z.\vec{y}_1]$, $f_{obs2} = [\vec{E}'_2(z')/z'.\vec{y}'_2]$. *If* m *is not synchronized,* $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ *in* comm *is dropped.*

(b) $\text{CALL}_{monitor}$*: For* $m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$, comm *is given by* $E_0(z) = z' \wedge \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$.

(c) CALL_{start}*: For* $m = \mathsf{start}$, comm *is* $E_0(z) = z' \wedge \neg z'.\mathsf{started}$, *where* $\{q_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{q_3\}\ stm; \mathsf{return}$ *is the body of the* run*-method of* c'.

(d) $\text{CALL}^{skip}_{start}$*: For* $m = \mathsf{start}$*, additionally, (33) must hold with* comm *given by* $E_0(z) = z' \wedge z'.\mathsf{started}$, $q_2 = q_3 = \mathsf{true}$, *and* f_{comm} *and* f_{obs2} *are the identity functions.*

2. (a) RETURN*: For all method call statements* $u_{ret} := e_0.m(\vec{e})\ \langle \vec{y}_1 := \vec{e}_1\rangle^{!call}\{p_1\}^{wait}\{p_2\}^{?ret}\langle \vec{y}_4 := \vec{e}_4\rangle^{?ret}\{p_3\}$ *(or such without receiving a value) occurring in* c *with* e_0 *of type* c'*, such that method* $m(\vec{u})$ *of* c' *has the return statement* $\{q_1\}\ \mathsf{return}\ e_{ret}\ \{q_2\}^{!ret}\langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\{q_3\}$*, Equation (33) must hold with* comm *given by* $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$*, and where* $f_{comm} = [E'_{ret}(z')/u_{ret}]$, $f_{obs1} = [\vec{E}'_3(z')/z'.\vec{y}'_3]$*, and* $f_{obs2} = [\vec{E}_4(z)/z.\vec{y}_4][\mathsf{null}/\mathsf{top}]$.

(b) RETURN_{wait}*: For* $\{q_1\}\,\mathsf{return}_{getlock}\ \{q_2\}^{!ret}\langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\{q_3\}$ *in a* wait*-method,* comm *is* $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge z'.\mathsf{lock} = \mathsf{free} \wedge \mathsf{thread}' \in z'.\mathsf{notified}$.

(c) RETURN_{run}*: For* $\{q_1\}\ \mathsf{return}\ \{q_2\}^{!ret}\langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\{q_3\}$ *occurring in a* run*-method,* $p_1 = p_2 = p_3 = \mathsf{true}$, $\mathsf{comm} = \mathsf{true}$*, and furthermore* f_{comm} *and* f_{obs2} *the identity function.*

Definition 17 (Cooperation test: Instantiation). *A proof outline satisfies the* cooperation test for object creation, *if for all classes* c' *and statements* $\{p_1\}\,u := \mathsf{new}^c\ \{p_2\}^{new}\langle \vec{y} := \vec{e}\rangle^{new}\{p_3\}$ *in* c'*:*

$$\models_{\mathcal{G}} z{\neq}\mathsf{null} \wedge z{\neq}u \wedge \exists z'.\ \big(\mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u.\ P_1(z)) \downarrow z'\big) \rightarrow P_2(z) \wedge I_c(u) \wedge (GI \wedge P_3(z)) \circ f_{obs}\ , \tag{34}$$

with $z \in LVar^{c'}$ *and* $z' \in LVar^{\mathsf{list\ Object}}$ *fresh , and where* $f_{obs} = [\vec{E}(z)/z.\vec{y}]$.

Definition 18 (Cooperation test: Exception handling). *A proof outline satisfies the* cooperation test for exception handling, *if for all statements* $u_{ret} := e_0.m(\vec{e})\ \langle stm\rangle^{!call}\{p_1\}^{wait}\{p_2\}^{?ret}\langle \vec{y}_4 := \vec{e}_4\rangle^{?ret}\{p_3\}$ *(or such without receiving a value) occurring in class* c *with* $m \neq \mathsf{start}$ *and* e_0 *of type* c'*, and for all* $\{q_1\}\ \mathsf{throw}\ e\,\{q_2\}^{throw}\langle \vec{y}_3 := \vec{e}_3\rangle^{throw}$ *in* $m(\vec{u})$ *of* c' *which is not in the try-block of any try-catch-finally statement,*

$$\models_{\mathcal{G}} GI \wedge P_1(z) \wedge Q'_1(z') \wedge \mathsf{comm} \rightarrow (P_2(z) \wedge Q'_2(z')) \circ f_{throw} \wedge (GI \wedge P_3(z)) \circ f_{obs2} \circ f_{obs1} \circ f_{throw}$$

must hold with distinct fresh logical variables $z \in LVar^c$ *and* $z' \in LVar^{c'}$*, and with* comm *given by* $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge E'(z') \neq \mathsf{null} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}$*. Furthermore,* f_{throw} *is* $[E'(z')/\mathsf{top}]$, f_{obs1} *is* $[\vec{E}'_3(z')/z'.\vec{y}'_3]$*, and* f_{obs2} *is* $[\vec{E}_4(z)/z.\vec{y}_4]$*. Rethrowing outside try-catch-finally blocks in run methods is similar.*

7 Soundness and completeness

This section contains soundness and completeness of the proof method of Section 6.2. Given a program together with its annotation, the proof system stipulates a number of induction conditions for the various types of assertions and program constructs. *Soundness* of the proof system means that for a proof outline satisfying the verification conditions, all configurations reachable in the operational semantics satisfy the given assertions. *Completeness* conversely means that if a program does satisfy an annotation, this fact is provable. For convenience, let us introduce the following notations: Given a program *prog*, we will write φ_{prog} or just φ for its annotation, and write $prog \models \varphi$, if *prog* satisfies all requirements stated in the assertions, and $prog' \vdash \varphi'$, if $prog'$ with annotation φ' satisfies the verification conditions of the proof system:

Definition 19. *Given a program prog with annotation φ, then $prog \models \varphi$ iff for all reachable configurations $\langle T, \sigma \rangle$ of prog, for all $(\alpha, \tau, stm) \in T$, and for all logical environments ω referring only to values existing in σ:*

1. $\omega, \sigma(\alpha), \tau \models_{\mathcal{L}} pre(stm)$, *and*
2. $\omega, \sigma \models_{\mathcal{G}} GI$.

Furthermore, for all classes c, objects $\beta \in Val^c(\sigma)$, and local states τ':

3. $\omega, \sigma(\beta), \tau' \models_{\mathcal{L}} I_c$.

For proof outlines, we write $prog' \vdash \varphi'$ iff $prog'$ with annotation φ' satisfies the verification conditions of the proof system.

In the following sections we discuss the basic ideas of the soundness and completeness proofs. The formal proofs can be found in the appendix.

7.1 Soundness

Soundness, as mentioned, means that all reachable configurations do satisfy their assertions for an annotated program that has been verified using the proof conditions. Soundness of the method is proved by a straightforward, albeit tedious, induction on the computation steps.

Before embarking upon the soundness formulation and its proof, we need to clarify the connection between the original program and the proof outline, i.e., the one extended by auxiliary variables, and decorated with assertions. The transformation is done for the sake of verification, only, and as far as the unaugmented portion of the states and the configurations is concerned, the behavior of the original and the transformed program are the same.

To make the connection between original program and the proof outline precise, we define a projection operation $\downarrow prog$, that jettisons all additions of the transformation. So let $prog'$ be a proof outline for *prog*, and $\langle T', \sigma' \rangle$ a global configuration of $prog'$. Then $\sigma' \downarrow prog$ is defined by removing all auxiliary instance variables from the instance state domains. For the set of thread configurations, $T' \downarrow prog$ is given by restricting the domains of the local states to non-auxiliary variables and removing all augmentations. Additionally, for local configurations $(\alpha, \tau, \mathsf{return}_{getlock}\langle stm\rangle^{!ret}) \in T'$, if the executing thread is in the wait set, i.e., if $(\tau(\mathsf{thread}), n) \in \sigma'(\alpha)(\mathsf{wait})$ for some n, then the statement $\mathsf{return}_{getlock}$ gets replaced by $?\mathsf{signal}; \mathsf{return}_{getlock}$. Furthermore, for local configurations $(\alpha, \tau, stm; \mathsf{return}\ \langle stm'\rangle^{!ret}) \in T'$ with $stm \neq \epsilon$ an auxiliary assignment in the notify- or the notifyAll-method, the auxiliary assignment stm gets replaced by !signal and !signal_all, respectively. The following lemma expresses that the transformation does not change the behavior of programs:

Lemma 5. *Let $prog'$ be a proof outline for a program $prog$. Then $\langle T, \sigma\rangle$ is a reachable configuration of $prog$ iff there exists a reachable configuration $\langle T', \sigma'\rangle$ of $prog'$ with $\langle T' \downarrow prog, \sigma' \downarrow prog\rangle = \langle T, \sigma\rangle$.*

The augmentation introduced a number of specific auxiliary variables that reflect the predicates used in the semantics. That the semantics is faithfully represented by the variables is formulated in the following lemmas.

Lemma 6 (Identification). *Let $\langle T, \sigma\rangle$ be a reachable configuration of a proof outline. Then*

1. *for all stacks ξ and ξ' in T and for all local configurations $(\alpha, \tau, stm) \in \xi$ and $(\alpha', \tau', stm') \in \xi'$ we have $\tau(\mathsf{thread}) = \tau'(\mathsf{thread})$ iff $\xi = \xi'$, and*
2. *for each stack $(\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n)$ in T and indices $0 \leq i, j \leq n$,*
   (a) $\tau_i(\mathsf{thread}) = \alpha_0$;
   (b) $i < j$ *and* $\alpha_i = \alpha_j$ *implies* $\tau_i(\mathsf{conf}) < \tau_j(\mathsf{conf}) < \sigma(\alpha_i)(\mathsf{counter})$,
   (c) $0 < j$ *implies* $\tau_j(\mathsf{caller}) = (\alpha_{j-1}, \tau_{j-1}(\mathsf{conf}), \tau_{j-1}(\mathsf{thread}))$, *and*
   (d) $\mathsf{proj}(\tau_0(\mathsf{caller}), 3) \neq \tau_0(\mathsf{thread})$,

where $proj(v, i)$ is the ith component of the tuple v.

Lemma 7 (Lock, Wait, Notify). *Let $\langle T, \sigma\rangle$ be a reachable configuration of a proof outline for the original program $prog$, $\alpha \in Val(\sigma)$ an object identity, and let $\xi = (\alpha_0, \tau_0, stm_0) \circ \xi' \in T$. Let furthermore n be the number synchronized method executions of ξ in α, i.e., $n = |\{(\alpha, \tau, stm) \in \xi \mid stm \text{ synchr.}\}|$. Then*

1. (a) $\neg owns(T \downarrow prog, \alpha)$ *iff* $\sigma(\alpha)(\mathsf{lock}) = free$
   (b) $owns(\xi \downarrow prog, \alpha)$ *iff* $\sigma(\alpha)(\mathsf{lock}) = (\alpha_0, n)$
2. (a) $\xi \in wait(T \downarrow prog, \alpha)$ *iff* $(\alpha_0, n) \in \sigma(\alpha)(\mathsf{wait})$
   (b) $\xi \in notified(T \downarrow prog, \alpha)$ *iff* $(\alpha_0, n) \in \sigma(\alpha)(\mathsf{notified})$
   (c) $proj(\sigma(\alpha)(\mathsf{wait})[i], 1) = proj(\sigma(\alpha)(\mathsf{wait})[j], 1)$ *implies* $i = j$
   (d) $proj(\sigma(\alpha)(\mathsf{notified})[i], 1) = proj(\sigma(\alpha)(\mathsf{notified})[j], 1)$ *implies* $i = j$
   (e) *if* $(\alpha_0, m) \in \sigma(\alpha)(\mathsf{wait})$ *or* $(\alpha_0, m) \in \sigma(\alpha)(\mathsf{notified})$ *then* $m = n$
   (f) $\sigma(\alpha)(\mathsf{wait}) \cap \sigma(\alpha)(\mathsf{notified}) = \emptyset$,

where $s[i]$ is the ith element of the sequence s.

The above Lemma assures disjunctness of the sequences stored in the wait and notified variables; if the order of the elements is unimportant, in the following we sometimes use set notation for their values.

Lemma 8 (Started). *For all reachable configurations $\langle T, \sigma\rangle$ of a proof outline for a program prog, and all objects $\alpha \in Val(\sigma)$, we have $started(T \downarrow prog, \alpha)$ iff $\sigma(\alpha)(\mathsf{started})$.*

Let *prog* be a program with annotation φ, and $prog'$ a a corresponding proof outline with annotation φ'. Let GI' be the global invariant of φ', I'_c denote its class invariants, and for an assertion p of φ let p' denote the assertion of φ' associated with the same control point. We write $\models \varphi' \to \varphi$ iff $\models_{\mathcal{G}} GI' \to GI$, $\models_{\mathcal{L}} I'_c \to I_c$ for all classes c, and $\models_{\mathcal{L}} p' \to p$, for all assertions p of φ associated with some control point. To give meaning to the auxiliary variables, the above implications are evaluated in the context of states of the augmented program. The following theorem states the soundness of the proof method.

Theorem 1 (Soundness). *Let $prog'$ be a proof outline with annotation $\varphi_{prog'}$.*

$$\textit{If} \quad prog' \vdash \varphi_{prog'} \quad \textit{then} \quad prog' \models \varphi_{prog'} \ .$$

The soundness proof is basically an induction on the length of computation, simultaneously on all three parts from Definition 19. For the inductive step, we assume that the verification conditions are satisfied and assume a reachable configuration satisfying the annotation. We make case distinction on the kind of the next computation step: If the computation step executes an assignment, then we use the local correctness conditions for inductivity of the executing local configuration's properties, and the interference freedom test for all other local configurations and the class invariants. For communication, invariance for the executing partners and the global invariant is shown using the cooperation test for communication. Exception handling and communication itself does not affect the global state; invariance of the remaining properties under the corresponding observations is shown again with the help of the interference freedom test. Finally for object creation, invariance for the global invariant, the creator local configuration, the created object's class invariant is assured by the conditions of the cooperation test for object creation; all other properties are shown to be invariant using the interference freedom test.

Theorem 1 is formulated for reachability of augmented programs. With the help of Lemma 5, we immediately get:

Corollary 1. *If $prog' \vdash \varphi_{prog'}$ and $\models \varphi_{prog'} \to \varphi_{prog}$, then $prog \models \varphi_{prog}$.*

7.2 Completeness

Next we conversely show that if a program satisfies the requirements asserted in its proof outline, then this is indeed provable, i.e., then there exists a proof outline which can be shown to hold and which implies the given one:

$$\forall prog.\ prog \models \varphi_{prog} \ \Rightarrow\ \exists prog'.\ prog' \vdash \varphi_{prog'} \wedge\ \models \varphi_{prog'} \to \varphi_{prog} \ .$$

Given a program satisfying an annotation $prog \models \varphi_{prog}$, the consequent can be uniformly shown, i.e., independently of the given assertional part φ_{prog}, by instantiating $\varphi_{prog'}$ to the strongest annotation still provable, thereby discharging the last clause $\models \varphi_{prog'} \rightarrow \varphi_{prog}$. Since the strongest annotation still satisfied by the program corresponds to reachability, the key to completeness is to

1. augment each program with enough information (see Definition 20 below), to be able to
2. express reachability in the annotation, i.e., annotate the program such that a configuration satisfies its local and global assertions exactly if reachable (see Definition 21 below), and finally
3. to show that this augmentation indeed satisfies the verification conditions.

We begin with the augmentation, using the transformation from Section 5.3 as starting point, where the programs are augmented with the specific auxiliary variables. To facilitate reasoning, we introduce an additional auxiliary local variable loc, which stores the current control point of the execution of a local configuration. Given a function which assigns to all control points unique location labels, we extend each assignment with the update $\mathsf{loc} := l$, where l is the label of the control point after the given occurrence of the assignment. Also unobserved statements are extended with the update. We write $l \equiv stm$ if l represents the control point in front of stm.

The standard way for completeness augmentation is to add information into the states about the way how it has been reached, i.e., the *history* of the computation leading to the configuration. This information is recorded using history variables.

The assertional language is split into a local and a global level, and likewise the proof system is tailored to separate local proof obligations from global ones to obtain a modular proof system. The history will be recorded in instance variables, and thus each instance can keep track only of its own past. To mirror the split into a local and a global level in the proof system, the history per instance is recorded separately for *internal* and *external* behavior. The sequence of internal state changes local to that instance is recorded in the *local* history and the external behavior in the *communication* history.

The local history keeps track of the state updates. We store in the local history the updated local and instance states of the executing local configuration and the object in which the execution takes place. Note that the local history stores also the values of the built-in auxiliary variables, and thus the identities of the executing thread and the executing local configuration.

The communication history keeps information about the kind of communication, the communicated values, and the identity of the communication partners involved. For the kind of communication, we distinguish as cases object creation, ingoing and outgoing method calls, and likewise ingoing and outgoing communication for the return value. We use the set $\bigcup_{c\in\mathcal{C}}\{\mathsf{new}^c\} \cup \bigcup_{m\in\mathcal{M}}\{!m, ?m\} \cup \{!\mathsf{return}, ?\mathsf{return}, !\,\mathsf{throw}, ?\,\mathsf{throw}\}$ of constants for this purpose, where $\mathcal{C}$ and $\mathcal{M}$ are the sets of all class and method names, respectively. Notification does not update the communication history, since it is object-internal computation. For the same reason, we don't record self-communication in h_{comm}. Note in passing that the information stored in the communication history matches exactly the information needed to decorate the transitions in order to obtain a compositional variant of the operational semantics of Section 4.2. See [ÁdBdRS03a] for such a compositional semantics.

Definition 20 (Augmentation with histories). *Every class is further extended by two auxiliary instance variables* h_{inst} *and* h_{comm}*, both initialized to the empty sequence. They are updated as follows:*

1. *Each multiple assignment* $\vec{y} := \vec{e}$ *in each class* c *that is not the observation of a method call or of the reception of a return value is extended with*

$$\mathsf{h}_{inst} := \mathsf{h}_{inst} \circ ((\vec{x}, \vec{v})[\vec{e}/\vec{y}]) \ ,$$

where $\vec{x}$ *are the instance variables of class* c *containing also* h_{comm} *but without* h_{inst}*, and* $\vec{v}$ *are the local variables. Observations* $\vec{y} := \vec{e}$ *of* $u_{ret} := e_0.m(\vec{e}')$ *and of the corresponding reception of the return value get extended with the assignment*

$$\mathsf{h}_{inst} := \mathsf{if}\ (e_0 = \mathsf{this})\ \mathsf{then}\ \mathsf{h}_{inst}\ \mathsf{else}\ \mathsf{h}_{inst} \circ ((\vec{x}, \vec{v})[\vec{e}/\vec{y}])\ \mathsf{fi}\,,$$

instead, if $m \neq \mathsf{start}$*. For* $e_0.\mathsf{start}(\vec{e}'); \langle \vec{y} := \vec{e} \rangle^{!call}$ *we use the same update with the condition* $e_0 = \mathsf{this}$ *replaced by* $e_0 = \mathsf{this} \wedge \neg\mathsf{started}$.
2. *Every observation of communication, object creation, or of a* throw *statement outside try-catch-finally blocks in a method different from* run *gets extended by*

$$\begin{aligned} \mathsf{h}_{comm} := {} & \mathsf{if}\ (\mathsf{partner} = \mathsf{this})\ \mathsf{then}\ \mathsf{h}_{comm}\ \mathsf{else} \\ & \mathsf{h}_{comm} \circ (\mathsf{sender}, \mathsf{receiver}, \mathsf{values})\ \mathsf{fi}\ , \end{aligned}$$

where the expressions partner, sender, receiver, *and* values *depend on the kind of communication as follows:*

communication	partner	sender	receiver	values
$u := \mathsf{new}^c$	null	this	null	$\mathsf{new}^c\ u$, thread
$u_{ret} := e_0.m(\vec{e})$	e_0	this	e_0	$!m(\vec{e})$
receive return	e_0	e_0	this	if top = null then ? return u_{ret}, thread else ? throw top, thread fi
receive call $m(\vec{u})$	caller_obj	caller_obj	this	$?m(\vec{u})$
return e_{ret}	caller_obj	this	caller_obj	! return e_{ret}, thread
throw e	caller_obj	this	caller_obj	! throw e, thread

with caller_obj *given by the first component of the variable* caller.

In the update of the history variable h_{inst}, the expression $(\vec{x}, \vec{u})[\vec{e}/\vec{y}]$ identifies the active thread and local configuration by the local variables thread and conf, and specifies its instance local state after the execution of the assignment. Note that especially the values of the auxiliary variables introduced in the augmentation are recorded in the local history. In the following we will also write (σ_{inst}, τ) when referring to elements of h_{inst}.

Note furthermore that the communication history records also the identities of the communicating threads in values.

Next we introduce the annotation for the augmented program.

Definition 21 (Reachability annotation). *We define the following annotation for the augmented program:*

1. $\omega, \sigma \models_{\mathcal{G}} GI$ *iff there exists a reachable* $\langle T, \sigma' \rangle$ *such that* $Val(\sigma) = Val(\sigma')$, *and for all* $\alpha \in Val(\sigma)$, $\sigma(\alpha)(\mathsf{h}_{comm}) = \sigma'(\alpha)(\mathsf{h}_{comm})$.
2. *For each class* c, *let* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} I_c$ *iff there is a reachable* $\langle T, \sigma \rangle$ *such that* $\sigma(\alpha) = \sigma_{inst}$, *where* $\alpha = \sigma_{inst}(\mathsf{this})$. *For each class* c *and method* m *of* c, *the pre- and postconditions of* m *are given by* I_c.
3. *For assertions at control points,* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} pre(stm)$ *iff there is a reachable* $\langle T, \sigma \rangle$ *with* $\sigma(\alpha) = \sigma_{inst}$ *for* $\alpha = \sigma_{inst}(\mathsf{this})$, *and* $(\alpha, \tau, stm; stm') \in T$.
4. *For preconditions* p *of observations observing a statement* stm *which is not an assignment, let* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ *iff there is a reachable* $\langle T, \sigma \rangle$ *with* $\sigma(\alpha) = \sigma_{inst}$ *for* $\alpha = \sigma_{inst}(\mathsf{this})$, *and with* $(\alpha, \tau', stm; stm') \in T$ *enabled to execute resulting in the local state* τ *directly after the execution of the statement but before its observation.*
   For observing the reception of a method call, instead of the existence of the enabled $(\alpha, \tau', stm; stm') \in T$, *we require that a call of the method of* α *is enabled in* $\langle T, \sigma \rangle$ *with resulting callee local state* τ *directly after communication*[19].

It can be shown that these assertions are expressible in the assertion language [TZ88]. The augmented program together with the above annotation build a proof outline that we denote by *prog'*.

What remains to be shown for completeness is that the proof outline *prog'* indeed satisfies the verification conditions of the proof system. Initial and local correctness are straightforward.

Completeness for the interference freedom test and the cooperation test are more complex, since, unlike initial and local correctness, the verification conditions in these cases mention more than one local configuration in their respective antecedents. Now, the reachability assertions of *prog'* guarantee that, when satisfied by an instance local state, there *exists* a reachable global configuration responsible for the satisfaction. So a crucial step in the completeness proof for

[19] For the precondition of the observation *stm* at the beginning of the run-method of the main class, $\langle T, \sigma \rangle$ can also be the initial configuration before the execution of the observation *stm*.

interference freedom and the cooperation test is to show that individual reachability of two local configurations implies that they are reachable in a *common* computation. This is also the key property for the history variables: they record enough information such that they allow to uniquely determine the way a configuration has been reached; in the case of instance history, uniqueness of course, only as far as the chosen instance is concerned. This property is stated formally in the following local merging lemma.

Lemma 9 (Local merging lemma). *Assume two reachable global configurations* $\langle T_1, \sigma_1 \rangle$ *and* $\langle T_2, \sigma_2 \rangle$ *of* $prog'$ *and* $(\alpha, \tau, stm) \in T_1$ *with* $\alpha \in Val(\sigma_1) \cap Val(\sigma_2)$*. Then* $\sigma_1(\alpha)(\mathsf{h}_{inst}) = \sigma_2(\alpha)(\mathsf{h}_{inst})$ *implies* $(\alpha, \tau, stm) \in T_2$*.*

For completeness of the cooperation test, connecting two possibly different instances, we need an analogous property for the communication histories. Arguing on the global level, the cooperation test can assume that two control points are individually reachable but agreeing on the communication histories of the objects. This information must be enough to ensure common reachability. Such a common computation can be constructed, since the internal computations of different objects are independent from each other, i.e., in a global computation, the local behavior of an object is interchangeable, as long as the external behavior does not change. This leads to the following lemma:

Lemma 10 (Global merging lemma). *Assume two reachable global configurations* $\langle T_1, \sigma_1 \rangle$ *and* $\langle T_2, \sigma_2 \rangle$ *of* $prog'$ *and* $\alpha \in Val(\sigma_1) \cap Val(\sigma_2)$ *with the property* $\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma_2(\alpha)(\mathsf{h}_{comm})$*. Then there exists a reachable configuration* $\langle T, \sigma \rangle$ *with* $Val(\sigma) = Val(\sigma_2)$*,* $\sigma(\alpha) = \sigma_1(\alpha)$*, and* $\sigma(\beta) = \sigma_2(\beta)$ *for all* $\beta \in Val(\sigma_2) \backslash \{\alpha\}$*.*

Note that together with the local merging lemma this implies that all local configurations in $\langle T_1, \sigma_1 \rangle$ executing in α and all local configurations in $\langle T_2, \sigma_2 \rangle$ executing in $\beta \neq \alpha$ are contained in the commonly reached configuration $\langle T, \sigma \rangle$.

This brings us to the completeness result:

Theorem 2 (Completeness). *For a program prog, the proof outline* $prog'$ *satisfies the verification conditions of the proof system from Section 6.2.*

8 Proving deadlock freedom

The previous sections described a proof system which can be used to prove safety properties of $Java_{synch}$ programs. In this section we show how to apply the proof system to prove *deadlock freedom.*

8.1 Expressing deadlock freedom

A system of processes is in a deadlocked configuration, if no one of them is enabled to compute, but not yet all processes are terminated. A typical deadlock situation can occur, if two threads t_1 and t_2 both try to gather the locks of two objects z_1 and z_2, but in reverse order: t_1 first applies for access to the synchronized methods of z_1, and then for those of z_2, while t_2 first collects the lock of z_2, and tries to become the lock owner of z_1. Now, it can happen, that t_1 gets the lock of z_1, t_2 gets the lock of z_2, and both are waiting for the other lock, which will never become free. Another typical source of deadlock situations are threads which suspended themselves by calling wait and which will never get notified.

So, what kind of statements can be disabled and under which conditions? The important cases, to which we restrict, are

- the invocation of synchronized methods, if the lock of the callee object is neither free nor owned by the executing thread,
- if a thread tries to invoke a monitor method of an object whose lock it doesn't own, or
- if a thread tries to return from a wait-method, but either the lock is not free or the thread is not yet notified.

To be exact, the semantics specifies method calls to be disabled also, if the callee object is the empty reference. However, we won't deal with this case; it can be excluded in the preconditions by stating that the callee object is not null.

Assume a proof outline with global invariant GI. For a logical variable z of type Object, let $I(z) = I[z/\mathsf{this}]$ be the class invariant of z expressed on the global level. Let the assertion $\mathsf{terminated}(z)$ express that the thread of z is already terminated. Formally, we define $\mathsf{terminated}(z)$ by $\exists\vec{v}.\, q[z/\mathsf{thread}][z/\mathsf{this}]$, where q is the postcondition of the run-method of z, and $\vec{v}$ its local variables. For assertions p in z' let furthermore $\mathsf{blocked}(z, z', p)$ express that the thread of z is disabled in the object z' at control point p. Formally, we define $\mathsf{blocked}(z, z', p)$ by

- $\exists\vec{v}.\, p[z/\mathsf{thread}][z'/\mathsf{this}] \wedge e_0.\mathsf{lock} \neq \mathsf{free} \wedge \mathsf{thread}(e_0.\mathsf{lock}) \neq \mathsf{thread}$ if p is the precondition of a call invoking a synchronized method of e_0,
- $\exists\vec{v}.\, p[z/\mathsf{thread}][z'/\mathsf{this}] \wedge \mathsf{thread}(e_0.\mathsf{lock}) \neq \mathsf{thread}$ if p is the precondition of a call invoking a monitor method of e_0,
- $\exists\vec{v}.\, p[z/\mathsf{thread}][z'/\mathsf{this}] \wedge (z'.\mathsf{lock} \neq \mathsf{free} \vee z \notin z'.\mathsf{notified})$ if p is the precondition of the return statement in the wait-method, and
- false otherwise,

where $\vec{v}$ is the vector of local variables in the given assertion, and z and z' fresh. Note that thread is substituted and thus the quantification over thread is without effect. Let finally $\mathsf{blocked}(z, z')$ express that the thread of object z is blocked in the object z'. It is defined by the assertion $\bigvee_{p \in Ass(z')} \mathsf{blocked}(z, z', p)$, where $Ass(z')$ is the set of all assertions in z'. Now we can formalize the verification condition for deadlock freedom:

Definition 22. *A proof outline satisfies the test for* deadlock freedom*, if*

$$
\begin{aligned}
\models_{\mathcal{G}} & (GI \wedge \\
& (\forall z.\, z \neq \mathsf{null} \rightarrow (I(z) \wedge \\
& \quad (z.\mathsf{started} \rightarrow (\mathsf{terminated}(z) \vee (\exists z'.\, z' \neq \mathsf{null} \wedge \mathsf{blocked}(z, z')))))) \wedge \\
& (\exists z.\, z \neq \mathsf{null} \wedge z.\mathsf{started} \wedge (\exists z'.\, z' \neq \mathsf{null} \wedge \mathsf{blocked}(z, z')))) \\
& \rightarrow \mathsf{false}\ .
\end{aligned} \tag{35}
$$

Soundness of the above condition, i.e., that the condition indeed assures absence of deadlock, is easy to show. Completeness results from the completeness of the proof method.

8.2 Deadlock freedom proof examples

For readability, we define the following functions, which describe properties of synchronization:

$$
\begin{aligned}
owns : & (\mathsf{Thread} \times (\mathsf{Thread} \times \mathsf{Int})) \rightarrow \mathsf{Bool}, \\
& owns(thread, lock) \stackrel{def}{=} thread \neq null \wedge proj(lock, 1) = thread \\
not\_owns : & (\mathsf{Thread} \times (\mathsf{Thread} \times \mathsf{Int})) \rightarrow \mathsf{Bool}, \\
& not\_owns(thread, lock) \stackrel{def}{=} thread \neq null \wedge proj(lock, 1) \neq thread \\
depth : & (\mathsf{Thread} \times \mathsf{Int}) \rightarrow \mathsf{Int}, \\
& depth(lock) \stackrel{def}{=} proj(lock, 2)
\end{aligned}
$$

The function *proj* is defined in Lemma 6; the *owns* function is already used in Example 15. In the following we apply the test for deadlock freedom to some examples. All examples are verified in *PVS*. The built-in augmentation is not listed in the code. We additionally list instance and local variable declarations `type name;`, where $\langle$`type name;`$\rangle$ declares auxiliary variables. We sometimes skip return statements without giving back a value, and write explicitly $\forall(z : t).p$ for quantification over t-typed values. All missing assertions are by definition true. An empty auxiliary observation $\langle\rangle$ in a notify- or notifyAll-method represents the built-in auxiliary assignment in the given method.

Reentrant monitors To demonstrate the basic idea of proving absence of deadlock, we first define a simple program, which does the following: The initial object, an instance of class `Main`, creates an instance of class `Synch`, starts its thread, and calls its synchronized `m1` method. The thread of the created instance also invokes `m1`, which simply calls the synchronized method `m2` of itself. Since synchronized methods cannot be executed simultaneously by different threads, either the initial thread or the thread of the new object calls `m1`, and then `m2`. The other thread has to wait until control returns from `m1`, before it can execute the invocations. The program is deadlock free, since *Java*'s monitor concept is reentrant, i.e., a thread owning the lock of an object may invoke several synchronized methods of that object.

Appendix D.1 contains a proof outline which satisfies the verification conditions and which implies the following invariant program properties:

```
class Main{
    ⟨ boolean in_Synch; ⟩
    ⟨ Synch created; ⟩

    nsync Void run(){
        Synch obj;
        obj := new^Synch; ⟨created = obj⟩^new
        obj.start();
        {(¬in_Synch) ∧ created = obj ∧ thread = this ∧ obj ≠ null ∧ obj ≠ this}
        obj.m1() ⟨in_Synch = (if obj = this then in_Synch else true fi)⟩^!call
                 ⟨in_Synch = (if obj = this then in_Synch else false fi)⟩^?ret
        {¬in_Synch}
    }
}

class Synch{
    sync Void m1(){
        {owns(thread, lock)}
        m2()
    }

    sync Void m2(){}

    nsync Void run(){
        {not_owns(thread, lock) ∧ thread = this ∧ started}
        m1()
        {not_owns(thread, lock)}
    }
}
```

with global invariant

$$
\begin{aligned}
&GI \stackrel{def}{=} \\
&(\forall(z : Synch).\, z \neq null \rightarrow (z.lock = (null, 0) \vee \\
&\qquad (\exists(t : Main).\, owns(t, z.lock) \wedge t.started \wedge t.created = z) \vee \\
&\qquad (owns(z, z.lock) \wedge z.started))) \wedge \\
&(\forall(t : Main).\, (t \neq null \wedge (\neg t.in\_Synch)) \rightarrow (t.created = null \vee not\_owns(t, t.created.lock))) \wedge \\
&(\forall(t : Main).\, t \neq null \rightarrow (\forall(z : Synch).\, (z \neq null \wedge owns(t, z.lock)) \rightarrow t.created = z))\,.
\end{aligned}
$$

The annotation shows properties at control points with terminated or possibly disabled execution, and implies, that a disabled or terminated thread owns the lock of a `Synch`-instance only if its current control point is in a synchronized method of the object. For threads of `Main`-instances this property cannot be expressed locally, thus we use the boolean auxiliary instance variable `in_Synch` to remember if the control point of the thread of the `Main`-instance is in itself or in the `Synch`-instance `obj`. To be able to refer to the identity of `obj` in the global language, we store the same identity in the auxiliary instance variable `created`. The global invariant GI combines properties of `Main`- and `Synch`-instances, stating that the lock of `Synch`-instances is either free, or owned by the creator of the instance or by the instance itself. Furthermore, if the variable `in_Synch` of a `Main`-instance z has the value *false*, than the thread of z does not hold the lock of z.created; `Main`-instances can own only the lock of the `Synch`-instance which they have been created.

The condition for deadlock freedom implies that there is an object $z \neq$ null whose thread is already started and whose execution is disabled in another object

$z' \neq \mathsf{null}$, i.e., $\mathsf{blocked}(z, z')$. First assume that z' is a `Main`-instance. Then the assertion $\mathsf{blocked}(z, z')$ implies that $z = z'$ is of type `Main`, and the thread of z tries to invoke method `m1` of $z'.\mathsf{created} \neq \mathsf{null}$, where the lock of $z'.\mathsf{created}$ is neither free nor owned by z, and we have $\neg z'.\mathsf{in\_Synch}$. Using the global invariant we get that there is an already started thread which owns the lock of $z'.\mathsf{created}$.

The antecedent of the deadlock freedom condition assures, that the execution of the lock owner is either disabled, or terminated. Let the current control point of the lock owner be in an object z''. This object cannot be a `Main`-instance: The assertions at both possible control points imply that the executing thread is the thread of z'' and that $\neg z''.\mathsf{in\_Synch}$. With the global invariant we get on the one hand $z''.\mathsf{created} = \mathsf{null} \vee \mathsf{not\_owns}(z'', z''.\mathsf{created.lock})$, and on the other hand GI states that the lock of $z'.\mathsf{created}$ can be owned by the object itself or by its creator, i.e., the assumption $\mathsf{owns}(z'', z'.\mathsf{created.lock})$ implies $z''.\mathsf{created} = z'.\mathsf{created}$, which leads to a contradiction. Thus the lock owner executes in a `Synch`-instance. We have three possible control points of the lock owner:

- The first possibility, prior to the invocation of `m2` in `m1` of z'', directly leads to a contradiction by the definition of the assertion $\mathsf{blocked}$: The precondition of the invocation states that the thread does own the lock of z'', and $\mathsf{blocked}$ extends this assertion by the assumption that the execution is not enabled, i.e., that the thread does not own the given lock.
- In the second case the lock owner is about to invoke `m1` in the `run`-method of z''. From the precondition of the invocation we get that the executing thread is the thread of z''. The global invariant implies that `Synch`-instances cannot own the lock of other `Synch`-instances. Now, by assumption z'' owns the lock of $z'.\mathsf{created}$, and with the above observation we get that $z'' = z'.\mathsf{created}$, i.e., z'' owns its own lock. But the precondition of the invocation implies, that the thread does not own the lock of z'', which leads to a contradiction.
- In the third case, the lock owner is the thread of z'' and is terminated. Again, the assumption that the executing thread, i.e., z'', owns the lock of $z'.\mathsf{created}$ implies with GI that $z'' = z'.\mathsf{created}$, i.e., that z'' owns its own lock. But the precondition implies that z'' does not own its own lock, which leads to a contradiction.

For the case that z' is a `Synch`-instance, we get from $\mathsf{blocked}(z, z')$ that the lock of z' is not free, but z is not the owner. The global invariant implies again, that there is an object whose thread is started and owns the lock of z'. The rest is analogous to the above case, where $z'.\mathsf{created}$ gets replaced by z'.

A simple wait-notify example Now let's have a look at an example demonstrating deadlock freedom for a notification process. Assume a program which defines two classes: The initial instance of the main class `Main` creates an instance of the class `Monitor`, and invokes its synchronized method `m1`, which starts its thread, and suspends the executing thread, thereby giving the lock free. Now the thread of the `Monitor`-instance can execute the synchronized method `m2`, probably producing some results which the other thread is waiting for. After the computation is completed, the lock owner sends a notification, and returns from `m2`. Now the other thread can continue its execution and use the produced data.

Again, Appendix D.2 lists a proof outline, which satisfies the verification conditions, and which implies the following invariant program properties:

$$
\begin{aligned}
&GI \stackrel{def}{=} \\
&\quad (\forall(z_1, z_2 : Main).(z_1 \neq null \wedge z_2 \neq null) \rightarrow z_1 = z_2) \wedge \\
&\quad (\forall(z_1, z_2 : Monitor).(z_1 \neq null \wedge z_2 \neq null) \rightarrow z_1 = z_2) \wedge \\
&\quad (\forall(z : Main).z \neq null \rightarrow (\\
&\qquad z.started \wedge \\
&\qquad (z.x = 1 \rightarrow (z.created \neq null \wedge z.created.lock = (null, 0))) \wedge \\
&\qquad (z.x = 3 \rightarrow (z.created \neq null \wedge z.created.x = 8)))) \wedge \\
&\quad (\forall(z_1 : Main).z_1 \neq null \rightarrow (\forall(z_2 : Monitor).(z_2 \neq null \wedge owns(z_1, z_2.lock)) \rightarrow z_2 = z_1.created)) \wedge \\
&\quad (\forall(z_1 : Monitor).z_1 \neq null \rightarrow (\forall(z_2 : Monitor).(z_2 \neq null \wedge owns(z_1, z_2.lock)) \rightarrow (z_1.started \wedge z_2 = z_1)))
\end{aligned}
$$

$$
\begin{aligned}
&I_{Monitor} \stackrel{def}{=} \\
&\quad ((x = 2 \vee x = 7) \rightarrow (lock = (creator, 1) \wedge started)) \wedge \\
&\quad ((x = 4 \vee x = 5) \rightarrow (lock = (this, 1) \wedge started)) \wedge \\
&\quad (x = 6 \rightarrow (lock = (null, 0) \wedge creator \in notified \wedge started))) \wedge \\
&\quad ((x = 3 \vee x = 8) \rightarrow lock = (null, 0) \wedge started)
\end{aligned}
$$

```
class Main{
    ⟨ int x; ⟩
    ⟨ Monitor created; ⟩

    nsync Void run(){
        Monitor obj;
        obj = new^Monitor; ⟨created = obj; x = 1⟩^new
        {x = 1 ∧ thread = this ∧ created = obj ∧ obj ≠ null}
        obj.m1() ⟨x = (if obj = this then x else 2 fi)⟩^!call
                 ⟨x = (if obj = this then x else 3 fi)⟩^?ret
        {x = 3}
    }
}

class Monitor{
    ⟨ Main creator; ⟩
    ⟨ int x; ⟩

    nsync Void wait(){
        ⟨x = 3⟩^?call
        {3 ≤ x ∧ x ≤ 6 ∧ thread = creator}
        return_getlock ⟨x = 7⟩^!ret

    }

    nsync Void notify(){ ⟨⟩ return ⟨x = 5⟩^!ret }

    sync Void m1(){
        ⟨creator = thread; x = 1⟩^?call
        start();
        {x = 2 ∧ thread = creator}
        wait();
        return ⟨x = 8⟩^!ret
    }

    nsync Void run(){
        ⟨x = 2⟩^?call
        {(x = 2 ∨ x = 3) ∧ thread = this}
        m2()
        {x = 6 ∨ x = 7 ∨ x = 8}
    }

    sync Void m2(){
        ⟨x = 4⟩^?call
        {x = 4 ∧ thread = this}
```

```
        notify();
        return ⟨x = 6⟩^{!ret}
    }
}
```

Note that the precondition of the method invocation in the `run`-method of `Main` together with the global invariant implies that the lock of the callee is free, i.e., threads cannot be blocked at this control point. Furthermore, the preconditions of both monitor method calls in `Monitor` imply with the class invariant that the executing thread owns the lock, i.e., also at these control points execution is always enabled.

We start again with the assumption that there is an object z whose thread is started but not yet terminated, and whose execution is disabled in the object z', where both z and z' are different from the empty reference. The object z can be an instance of one of the classes `Main` or `Monitor`. According to the above observations, z' must be an instance of `Monitor`, and the control point is in the `wait`-method or prior to the invocation of `m2` in the `run`-method.

In the first case, the local assertion attached to the control point in the `wait`-method implies that $z = z'.\mathsf{creator}$, an instance of `Main`, does not own the lock of z' and that the thread of z' is started. Due to the assumptions of the deadlock freedom condition, the execution of the thread of z' is disabled or terminated. However, using the annotation, termination would imply $z'.x = 6$ and by the class invariant the execution of the thread of z would be enabled. The thread of z' can neither be in the `wait`-method, because the local assertion there implying $\mathsf{thread} = \mathsf{creator}$ would lead to a type contradiction. Thus the thread of z' executes the `run`-method of z', and is going to invoke the synchronized method `m2`. Since $z = z'.\mathsf{creator}$ does not own the lock of z' by assumption, the precondition of the invocation and the class invariant imply that the lock is free, and thus that the execution of z' is enabled.

The second case, when the thread of z is in the `run`-method of z' prior to the call of `m2`, is similar.

A producer-consumer example The proof outline below defines two classes `Producer` and `Consumer`, where `Producer` is the main class. The initial thread of the initial `Producer`-instance creates a `Consumer`-instance and calls its synchronized `produce` method. This method starts the consumer thread and enters a non-terminating loop, producing some results, notifying the consumer, and suspending itself by calling `wait`. After the producer suspended itself, the consumer thread calls the synchronized `consume` method, which consumes the result of the producer, notifies, and calls `wait`, again in a non-terminating loop.

Again, we only list a partial annotation and augmentation, which already implies deadlock freedom; see Appendix D.3 for the complete inductive proof outline.

$$
\begin{aligned}
&GI \stackrel{def}{=}\\
&\quad (\forall(p: Producer).(p \neq null \wedge \neg p.outside \wedge p.consumer \neq null) \rightarrow p.consumer.lock = (null, 0))\wedge\\
&\quad (\forall(c: consumer).(c \neq null \wedge c.started) \rightarrow (c.producer \neq null \wedge c.producer.started))\wedge
\end{aligned}
$$

$$(\forall(c1 : consumer).(c1 \neq null \rightarrow (\forall(c2 : consumer).c2 \neq null \rightarrow c1 = c2))$$

$$I_{Producer} \stackrel{def}{=} true$$

$$I_{Consumer} \stackrel{def}{=} (lock = (null, 0) \vee (owns(this, lock) \wedge started) \vee owns(producer, lock)) \wedge length(wait) \leq 1$$

```
class Producer {
    ⟨ Consumer consumer; ⟩
    ⟨ boolean outside; ⟩

    nsync Void wait(){ {false} }

    nsync Void run(){
        Consumer c;
        c = new^Consumer; ⟨consumer = c⟩^new
        {c = consumer ∧ ¬outside ∧ consumer ≠ null ∧ consumer ≠ this ∧ thread = this}
        c.produce(); ⟨outside = (if c = this then outside else true fi)⟩^!call
        {false}
    }
}

class Consumer {
    int buffer;
    ⟨ Producer producer; ⟩

    nsync Void wait(){
        {started ∧ not_owns(thread, lock) ∧ (thread = this ∨ thread = producer)∧
            (thread ∈ wait ∨ thread ∈ notified)}
    }

    sync Void produce(){
        int i;

        ⟨producer = proj(caller, 1)⟩^?call
        i=0;
        start();
        while (true){
            //produce i here
            buffer = i;
            {owns(thread, lock)}
            notify();
            {owns(thread, lock)}
            wait()
        }
    }

    nsync Void run(){
        {not_owns(thread, lock) ∧ thread = this}
        consume();
        {false}
    }

    sync Void consume(){
        int i;

        while (true){
            i = buffer;
            //consume i here
            {owns(thread, lock)}
            notify();
            {owns(thread, lock)}
            wait()
        }
    }
}
```

Both `run`-methods have false as postcondition, stating that the corresponding threads don't terminate. The preconditions of all monitor method invocations express that the executing thread owns the lock, and thus execution cannot be enabled at these control points. The `wait`-method of `Producer`-instances is not invoked; we define false as the precondition of its return statement, implying that disabledness is excluded also at this control point.

The condition for deadlock freedom assumes that there is a thread which is started but not yet terminated, and whose execution is disabled. This thread is either the thread of a `Producer`-instance, or that of a `Consumer`-instance.

We discuss only the case that the disabled thread belongs to a `Producer`-instance z different from the empty reference; the other case is similar. Note that the control of the thread of z cannot stay in the `run`-method of a `Consumer`-instance, since the corresponding local assertion implies $\mathsf{thread} = \mathsf{this}$, which would contradict to the type assumptions. Thus the thread can have its control point prior to the method call in the `run`-method of a `Producer`-instance, or in the `wait`-method of a `Consumer`-instance. In the first case, the corresponding local assertion and the global invariant imply that the lock of the callee is free, i.e., that the execution is enabled, which is a contradiction. In the second case, if the thread of z executes in the `wait`-method of a `Consumer`-instance z', the local assertion in `wait` together with the type assumptions implies $z'.\mathsf{started} \wedge \mathsf{not\_owns}(z, z'.\mathsf{lock}) \wedge z = z'.\mathsf{producer}$, and that z is either in the wait- or in the notified-set of z'.

According to the assumptions of the deadlock freedom condition, also the started thread of z' is disabled or terminated; its control point cannot be in a `Producer`-instance, since that would contradict to the type assumptions. Thus the control of z' stays in the `run`- or in the `wait`-method of a `Consumer`-instance; the annotation implies that the instance is z' itself.

If the control stays in the `run`-method, then the corresponding local assertion and the class invariant imply that the lock is free, since neither the producer, nor the consumer owns it, which leads to a contradiction, since in this case the execution of the thread of z' would be enabled. Finally, if the control of the thread of z' stays in the `wait`-method of z', then the annotation assures that the thread doesn't own the lock of z'; again, using the class invariant we get that the lock is free.

Now, both threads of z and z' have their control points in the `wait`-method of z', and the lock of z' is free. Furthermore, both threads are disabled, and are in the wait- or in the notified set. If one of them is in the notified set, than its execution is enabled, which is a contradiction. If both threads are in the wait set, then from $z \neq z'$ we imply that the wait-set of z' has at least two elements, which contradicts to the class invariant of z'.

Thus the assumptions lead to a contradiction, which was to show.

9 Conclusion

In this work we presented a tool-supported assertional proof method for a *Java* sublanguage including multithreading and exception handling. We introduced the language and the proof system incrementally in four steps: We started with a *sequential Java* sublanguage and its proof system. In the next step we included dynamic thread creation, resulting in a *multithreaded* sublanguage. In the next staged we extended the language and the proof system to cover *monitor synchronization* and *exception handling*. We gave proofs of soundness and completeness. The proof system supports also showing absence of deadlock.

We illustrated the use of our assertional proof system on a small number of examples, which have been verified using the tool *Verger*. The tool takes an augmented and annotated *Java* program, a so-called proof outline, as input and generates the verification conditions, which assure invariance of the annotation. We used the theorem prover *PVS* to verify the conditions.

Future work The preceding sections on possible extensions and on related work show, that there are a lot of challenging and interesting research topics in the field, which need further analysis. The incremental development illustrated how to extend the language and the proof system to deal with additional language features. As to future work, we plan to extend the programming language by further constructs, like inheritance and subtyping. We are also interested on the development of a compositional proof system. Also further development of the tool is of interest.

References

[AdB93] Pierre America and Frank S. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1993.

[ÁdBdRS03a] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A compositional operational semantics for Java$_{MT}$. In Nachum Derschowitz, editor, *International Symposium on Verification (Theory and Practice)*, volume 2772 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003. To appear. A preliminary version appeared as Technical Report TR-ST-02-2, May 2002.

[ÁdBdRS03b] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Inductive proof-outlines for monitors in Java. In Nestmann and Stevens [NS03]. To appear. A longer version appeared as Software Technologie technical report TR-ST-03-1, April 2003.

[ÁdBdRS03c] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A tool-supported assertional proof system for multithreaded Java. In Susan Eisenbach, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter, and Erik Poll, editors, *Proc. of the Workshop on Formal Techniques for Java-like Programs - FTfJP'2003*, 2003.

[AF99] Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS State-of-the-Art-Survey*. Springer-Verlag, 1999.

[AFdR80] Krzysztof R. Apt, Nissim Francez, and Willem-Paul de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.

[AJ01] Isabelle Attali and Thomas Jensen, editors. *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, 2001.

[ÁMdB00] Erika Ábrahám-Mumm and Frank S. de Boer. Proof-outlines for threads in Java. In Catuscia Palamidessi, editor, *Proceedings of CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2000.

[ÁMdBdRS01] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Deductive verification for multithreaded java (extended abstract). In *Proceedings of the Kolloquium Programmiersprachen und Grundlagen der Programmierung, 2001, Rurberg*, pages 121–126, 2001.

[ÁMdBdRS02a] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A tool-supported proof system for monitors in Java. In *Proceedings of the FMCO 2002*, 2002. To appear.

[ÁMdBdRS02b] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept. In Mogens Nielsen and Uffe H. Engberg, editors, *Proceedings of FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 4–20. Springer-Verlag, April 2002. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[Bal03] The project Bali. `http://isabelle.in.tum.de/Bali/`, 2003.

[BdBdRG03] Marcello Bosangue, Frank S. de Boer, Willem-Paul de Roever, and Susanne Graf, editors. *Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002), Leiden*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[CKRW99] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [AF99], pages 157–200.

[dB99] Frank S. de Boer. A WP-calculus for OO. In Wolfgang Thomas, editor, *Proceedings of FoSSaCS '99*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–156. Springer-Verlag, 1999.

[dBP02] Frank S. de Boer and Cees Pierik. Computer-aided specification and verification of annotated object-oriented programs. In Bart Jacobs and Arend Rensink, editors, *Proceedings of the Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, volume 209, pages 163–177. Kluwer, 2002.

[dBP03] Frank S. de Boer and Cees Pierik. Towards an environment for the verification of annotated object-oriented programs. Technical report UU-CS-2003-002, Institute of Information and Computing Sciences, University of Utrecht, January 2003.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.

[HJ89] Charles A. R. Hoare and Cliff B. Jones, editors. *Essays in Computing Science*. International Series in Computer Science. Prentice Hall, 1989.

[HJ00] Marieke Huisman and Bart Jacobs. Inheritance in higher order logic: Modeling and reasoning. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics (TPHOL 2000)*, volume 1869 of *Lecture Notes in Computer Science*, pages 301–319. An earlier version appeared as Technical Report CSI-R0004 Computing Science Institute, University of Nijmegen., 2000.

[HJvdB01] Marieke Huisman, Bart Jacobs, and Joachim van den Berg. A case study in class library verification: Java's vector class. *Software Tools for Technology Transfer*, 3(3):332–352, 2001.

[Hoa69] Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [HJ89].

[Hui01] Marieke Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.

[Hus01] Heinrich Hussmann, editor. *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science.* Springer-Verlag, 2001.

[Jac01] Bart Jacobs. A formalisation of Java's exception mechanism. In Sands [San01], pages 284–301.

[JKW03] Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java program verification challenges. In Bosangue et al. [BdBdRG03].

[JP01] Bart Jacobs and Eric Poll. A logic for the Java Modelling Language JML. In Hussmann [Hus01], pages 284–299.

[JvdBH+98a] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Barkum, Ulrich Hensel, and Hendrik Tews. Reasoning about classes in Java (preliminary report). In OOPSLA'98 [OOP98], pages 329–340. In *SIGPLAN Notices* 30(10).

[JvdBH+98b] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrick Tews. Reasoning about Java classes. In *Proceedings, Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 329–340, Vancouver, Canada, 1998.

[LG81] Gary Levin and David Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.

[Loo01] The LOOP project: Formal methods for object-oriented systems. http://www.cs.kun.nl/~bart/LOOP/, 2001.

[Nip02] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[NS03] Uwe Nestmann and Perdita Stevens, editors. *Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS '03), Paris*, Lecture Notes in Computer Science. Springer-Verlag, November 2003. To appear.

[NvO98] Tobias Nipkow and David von Oheimb. Java-light is type-safe — definitely. In POPL'98 [POP98], pages 161–170.

[NvOP00] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[OOP98] ACM. *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA) '98*, 1998. In *SIGPLAN Notices* 30(10).

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer-Verlag, 1992.

[Pau93] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993.

[PdB03] Cees Pierik and Frank S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In Nestmann and Stevens [NS03]. A extended version appeared as University of Utrecht Technical Report UU-CS-2003-010.

[PH97a] Arnd Poetzsch-Heffter. A logic for the verification of object-oriented programs. In Rudolf Berghammer and Friedeman Simon, editors, *Proceedings of Programming Languages and Fundamentals of Programming*, pages 31–42. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, November 1997. Bericht Nr. 9717.

[PH97b] Arnd Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* Technische Universität München, January 1997. Habilitationsschrift.

[PHM98] Arnd Poetzsch-Heffter and Peter Müller. Logical foundations for typed object-oriented languages. In David Gries and Willem-Paul de Roever, editors, *Proceedings of PROCOMET '98*. International Federation for Information Processing (IFIP), Chapman & Hall, 1998.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.

[POP98] ACM. *25th Annual Symposium on Principles of Programming Languages (POPL) (San Diego, CA)*, 1998.

[PvdBJ00] Eric Poll, Joachim van den Berg, and Bart Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 135–154. Kluwer Acad. Publ., 2000.

[PvdBJ01] Eric Poll, Joachim van den Berg, and Bart Jacobs. Formal specification of the Java Card API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.

[San01] David Sands, editor. *Programming Languages and Systems: Proceedings of the 10th European Symposium on Programming (ESOP 2001), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2001), (Genova, Italy, April 2001)*, volume 2028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

[TZ88] John V. Tucker and Jeffery I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*, volume 6 of *CWI Monograph Series*. North-Holland, 1988.

[vdBHJP00] Joachim van den Berg, Marieke Huisman, Bart Jacobs, and Eric Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P.D. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2000. A earlier version appeared as Computer Science Institute, University of Nijmegen, Technical Report CSI-R9926, 1999.

[vdBJ02] Joachim van den Berg and Bart Jacobs. The Loop compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems(TACAS' 02)*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer-Verlag, 2002.

[vdBJP01] Joachim van den Berg, Bart Jacobs, and Eric Poll. Formal specification and verification of JavaCard's application identifier class. In Attali and Jensen [AJ01], pages 137–150.

[vO00a] David von Oheimb. Axiomatic semantics for Java^{light}. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications European Conference on Object-Oriented Programming (OOPSLA) (ECOOP)*. 2000.

[vO00b] David von Oheimb. Axiomatic sematics for Java^{light} in Isabelle/HOL. Technical Report CSE 00-008, Oregon Graduate Institute, 2000.

[vO01] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[vON99] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Alves-Foss [AF99].

[vON02] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P.-A. Lindsay, editors, *Proceedings of Formal Methods Europe: Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 2002.

[WK99] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml.* Object Technology Series. Addison-Wesley, 1999.

A Proofs of properties of substitutions and projection

Proof (of Lemma 1). By induction on the structure of local expressions and assertions. The base cases for local expressions are listed below, where the ones for instance and local variables are covered by the respective provisos of the lemma.

$$
\begin{aligned}
[\![x[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![z.x]\!]_{\mathcal{G}}^{\omega,\sigma} = \sigma([\![z]\!]_{\mathcal{G}}^{\omega,\sigma})(x) = \sigma(\omega(z))(x) = [\![x]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} \\
[\![u[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![u]\!]_{\mathcal{G}}^{\omega,\sigma} = \omega(u) = \tau(u) = [\![u]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} \\
[\![\mathsf{this}[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![z]\!]_{\mathcal{G}}^{\omega,\sigma} = \omega(z) = [\![\mathsf{this}]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} \\
[\![\mathsf{null}[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= null = [\![\mathsf{null}]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} \\
[\![z'[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![z']\!]_{\mathcal{G}}^{\omega,\sigma} = \omega(z') = [\![z']\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} .
\end{aligned}
$$

Compound expressions are treated by straightforward induction:

$$
\begin{aligned}
& [\![\mathsf{f}(e_1,\ldots,e_n)[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} & \\
= \; & f([\![e_1[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma},\ldots,[\![e_n[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma}) & \text{semantics of assertions} \\
= \; & f([\![e_1]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau},\ldots,[\![e_n]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau}) & \text{by induction} \\
= \; & [\![\mathsf{f}(e_1,\ldots,e_n)]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} & \text{semantics of assertions .}
\end{aligned}
$$

For local assertions, negation and conjunction are straightforward. Unrestricted quantification $\exists z'.\ p$ in the local assertion language is only allowed for variables of type $t \in \{\mathsf{Int}, \mathsf{Bool}\}$ and for types composed from them, for which $Val^t_{null}(\sigma) = Val^t$. We get

$$
\begin{aligned}
& [\![(\exists z'.\ p)[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = true & \\
\iff \; & [\![\exists z'.\ p[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = true & \text{def. substitution} \\
\iff \; & [\![p[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega[z'\mapsto v],\sigma} = true \text{ for some } v \in Val^t & \text{assertion semantics} \\
\iff \; & [\![p]\!]_{\mathcal{L}}^{\omega[z'\mapsto v],\sigma(\omega(z)),\tau} = true \text{ for some } v \in Val^t & \text{by induction} \\
\iff \; & [\![\exists z'.\ p]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} = true & \text{assertion semantics.}
\end{aligned}
$$

For restricted quantification over elements of a sequence let $z' \in LVar^t$. Then

$$
\begin{aligned}
& [\![(\exists z' \in e.\ p)[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = true & \\
\iff \; & [\![\exists z'.\ z' \in e[z/\mathsf{this}] \wedge p[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = true & \text{by definition} \\
\iff \; & [\![z' \in e[z/\mathsf{this}] \wedge p[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega',\sigma} = true & \text{semantics} \\
& \text{for some } v \in Val^t_{null}(\sigma) \text{ and } \omega' = \omega[z' \mapsto v] & \\
\iff \; & \left([\![z']\!]_{\mathcal{G}}^{\omega',\sigma} \in [\![e[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega',\sigma} \wedge [\![p[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega',\sigma}\right) = true & \text{semantics} \\
& \text{for some } v \in Val^t_{null}(\sigma) \text{ and } \omega' = \omega[z' \mapsto v] & \\
\iff \; & \left([\![z']\!]_{\mathcal{L}}^{\omega',\sigma(\omega(z)),\tau} \in [\![e]\!]_{\mathcal{L}}^{\omega',\sigma(\omega(z)),\tau} \wedge [\![p]\!]_{\mathcal{L}}^{\omega',\sigma(\omega(z)),\tau}\right) = true & \text{by induction} \\
& \text{for some } v \in Val^t_{null}(\sigma) \text{ and } \omega' = \omega[z' \mapsto v] & \\
\iff \; & [\![(z' \in e) \wedge p]\!]_{\mathcal{L}}^{\omega',\sigma(\omega(z)),\tau} = true & \text{semantics} \\
& \text{for some } v \in Val^t_{null}(\sigma) \text{ and } \omega' = \omega[z' \mapsto v] & \\
\iff \; & [\![\exists z' \in e.\ p]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} = true & \text{semantics .}
\end{aligned}
$$

The last equation uses the assumption that the local state τ and the instance state $\sigma(\omega(z))$ assign values from $Val_{null}(\sigma)$ to all variables, i.e., e does not refer to values of non-existing objects (see Lemma 11). Consequently, $v \in Val^t_{null}$ together with $[\![z' \in e]\!]_{\mathcal{L}}^{\omega[z' \mapsto v],\sigma(\omega(z)),\tau} = true$ implies $v \in Val^t_{null}(\sigma)$. The case for restricted quantification over subsequences is analogous. □

Proof (of Lemma 3). We proceed by straightforward induction on the structure of local assertions. Let $\acute{\sigma}_{inst} = \grave{\sigma}_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}}]$ and $\acute{\tau} = \grave{\tau}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}}]$. In the case for local variables $u = y_i$ we get

$$\begin{aligned} [\![u[\vec{e}/\vec{y}]]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}} &= [\![e_i]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}} \\ &= \acute{\tau}(u) \\ &= [\![u]\!]_{\mathcal{L}}^{\omega,\acute{\sigma}_{inst},\acute{\tau}} . \end{aligned}$$

For instance variables $x = y_i$ similarly:

$$\begin{aligned} [\![x[\vec{e}/\vec{y}]]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}} &= [\![e_i]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}} \\ &= \acute{\sigma}_{inst}(x) \\ &= [\![x]\!]_{\mathcal{L}}^{\omega,\acute{\sigma}_{inst},\acute{\tau}} . \end{aligned}$$

The remaining cases are straightforward. □

Proof (of Lemma 4). Let $\acute{\omega} = \grave{\omega}[\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}]$ and $\acute{\sigma} = \grave{\sigma}[[\![z]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}.\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}]$. We proceed by induction on the structure of global expressions and assertions. The base cases for null and z' are straightforward. For the induction cases, we start with the crucial one for qualified reference to instance variables. For expressions $E'.x[\vec{E}/z.\vec{y}]$ with x not in $\vec{y}$ the property holds by induction. So assume that x is in $\vec{y}$:

$$[\![(E'.y_i)[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} = [\![\mathsf{if}\ E'[\vec{E}/z.\vec{y}] = z\ \mathsf{then}\ E_i\ \mathsf{else}\ (E'[\vec{E}/z.\vec{y}]).y_i\ \mathsf{fi}]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} .$$

This conditional assertion evaluates to $[\![E_i]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}$ if $[\![E'[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} = [\![z]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}$ and to $[\![(E'[\vec{E}/z.\vec{y}]).y_i]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}$ otherwise. So in the first case we get

$$\begin{aligned} [\![(E'.y_i)[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} &= [\![E_i]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} \\ &= \acute{\sigma}([\![z]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}})(y_i) && \text{by def. of } \acute{\sigma} \\ &= \acute{\sigma}([\![E'[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}})(y_i) && \text{by the case assumption} \\ &= \acute{\sigma}([\![E']\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}})(y_i) && \text{by induction} \\ &= [\![E'.y_i]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} && \text{by def. of } [\![\_]\!]_{\mathcal{G}} . \end{aligned}$$

If otherwise $[\![E'[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} \neq [\![z]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}$, then

$$\begin{aligned} [\![(E'.y_i)[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} &= [\![(E'[\vec{E}/z.\vec{y}]).y_i]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} \\ &= \grave{\sigma}([\![E'[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}})(y_i) && \text{by def. of } [\![\_]\!]_{\mathcal{G}} \\ &= \acute{\sigma}([\![E'[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}})(y_i) && \text{case assumption+def. } \acute{\sigma} \\ &= \acute{\sigma}([\![E']\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}})(y_i) && \text{by induction} \\ &= [\![E'.y_i]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} && \text{by def. of } [\![\_]\!]_{\mathcal{G}} . \end{aligned}$$

For operator expressions we get:

$$\begin{array}{lll} & [\![(\mathsf{f}(E_1,\ldots,E_n))[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} & \\ = & [\![\mathsf{f}(E_1[\vec{E}/z.\vec{y}],\ldots,E_n[\vec{E}/z.\vec{y}])]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} & \text{def. substitution} \\ = & f([\![E_1[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}},\ldots,[\![E_n[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}) & \text{def. } [\![\_]\!]_{\mathcal{G}} \\ = & f([\![E_1]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}},\ldots,[\![E_n]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}}) & \text{by induction} \\ = & [\![\mathsf{f}(E_1,\ldots,E_n)]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} & \text{def. } [\![\_]\!]_{\mathcal{G}}\ . \end{array}$$

For global assertions, the cases of negation and conjunction are straightforward. For quantification,

$$\begin{array}{lll} & [\![(\exists z'.\ P)[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} = \mathit{true} & \\ \iff & [\![\exists z'.\ P[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} = \mathit{true} & \text{def. substitution} \\ \iff & [\![P[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\grave{\omega}[z'\mapsto v],\grave{\sigma}} = \mathit{true} \text{ for some } v \in \mathit{Val}_{null}(\grave{\sigma}) & \text{def. } [\![\_]\!]_{\mathcal{G}} \\ \iff & [\![P]\!]_{\mathcal{G}}^{\acute{\omega}[z'\mapsto v],\acute{\sigma}} = \mathit{true} \text{ for some } v \in \mathit{Val}_{null}(\grave{\sigma}) & \text{by induction} \\ \iff & [\![\exists z'.\ P]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} = \mathit{true}\ , & \mathit{Val}(\grave{\sigma}) = \mathit{Val}(\acute{\sigma}) \end{array}$$

where z' is not in $\vec{y}$ (otherwise the substitution renames z'). □

Lemma 11. *Let σ be a global state and ω a logical environment referring only to values existing in σ. Then $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \in \mathit{Val}_{null}(\sigma)$ for all global expressions $E \in \mathit{GExp}$ that can be evaluated in the context of ω and σ.*

Proof (of Lemma 11). By structural induction on the global assertion. The case for logical variables $z \in \mathit{LVar}^t$ is immediate by the assumption about ω, the ones for null and operator expressions are trivial, respectively follows by induction. For qualified references $E.x$ with $E \in \mathit{GExp}^c$ and x an instance variable of type t in class c, if $E.x$ can be evaluated in the context of ω and σ, then $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \neq \mathit{null}$. Hence by induction $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \in \mathit{Val}_{null}(\sigma)$, more specifically $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \in \mathit{Val}(\sigma)$. Therefore by definition of global states $\sigma([\![E]\!]_{\mathcal{G}}^{\omega,\sigma})(x) \in \mathit{Val}_{null}(\sigma)$. □

Proof (of Lemma 2). We prove the lemma by structural induction on global assertions. Assume a global state $\grave{\sigma}$, and let $\acute{\sigma} = \grave{\sigma}[\alpha \mapsto \sigma_{inst}^{c,init}]$ be an extension of $\grave{\sigma}$ with a new object $\alpha \in \mathit{Val}^c$, $\alpha \notin \mathit{Val}(\grave{\sigma})$. Assume furthermore a logical environment ω referring only to values existing in $\grave{\sigma}$, and let v be the sequence consisting of all elements of $\bigcup_c \mathit{Val}_{null}^c(\grave{\sigma})$. Let finally P be a global assertion, $z' \in \mathit{LVar}^{\mathsf{list\ Object}}$ a logical variable not occurring in P, and $\acute{\omega} = \grave{\omega}[z' \mapsto v]$. Since z' is fresh in P, we have for all logical variables z in P that $[\![z]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} = \grave{\omega}(z) = \acute{\omega}(z) = [\![z]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} = [\![z \downarrow z']\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}}$. For qualified references to instance variables, the argument is as follows:

$$\begin{array}{lll} [\![E.x]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} & = \grave{\sigma}([\![E]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}})(x) & \text{semantics} \\ & = \acute{\sigma}([\![E]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}})(x) & [\![E]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} \neq \alpha \text{ by Lemma 11 and } \alpha \notin \mathit{Val}(\grave{\sigma}) \\ & = \acute{\sigma}([\![E \downarrow z']\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}})(x) & \text{by induction} \\ & = [\![(E \downarrow z').x]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} & \text{semantics} \\ & = [\![(E.x) \downarrow z']\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}} & \text{def. } \downarrow z'\ . \end{array}$$

The interesting case is the one for quantification. For $z \in LVar^t$:

$$
\begin{array}{lll}
 & \grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \exists z.\ P & \\
\iff & \grave{\omega}[z \mapsto u], \grave{\sigma} \models_{\mathcal{G}} P \text{ for some } u \in Val^t_{null}(\grave{\sigma}) & \text{semantics} \\
\iff & \acute{\omega}[z \mapsto u], \acute{\sigma} \models_{\mathcal{G}} P \downarrow z' \text{ for some } u \in Val^t_{null}(\grave{\sigma}) & \text{induction} \\
\iff & \acute{\omega}[z \mapsto u], \acute{\sigma} \models_{\mathcal{G}} \mathsf{obj}(z) \subseteq z' \wedge P \downarrow z' & obj(u) \subseteq v \\
 & \qquad \text{for some } u \in Val^t_{null}(\grave{\sigma}) & \\
\iff & \acute{\omega}, \acute{\sigma} \models_{\mathcal{G}} \exists z.\ \mathsf{obj}(z) \subseteq z' \wedge P \downarrow z' & \text{semantics} \\
\iff & \acute{\omega}, \acute{\sigma} \models_{\mathcal{G}} (\exists z.\ P) \downarrow z'. &
\end{array}
$$

The remaining cases are straightforward. □

B Soundness proof

This section contains the inductive proof of soundness of the proof method. We start with some ancillary lemmas about basic invariant properties of proof outlines, for instance properties of the built-in auxiliary variables added in the transformation. Afterwards, we show soundness of the proof system.

B.1 Invariant properties

Proof (of the transformation Lemma 5). We proceed for both directions by straightforward induction on the length of reduction. The only interesting property of the transformation is the representation of notification by a single auxiliary assignment of the notifier. For this case we use Lemma 7 showing soundness of the representation of the wait and notified sets by the auxiliary instance variables wait and notified. □

Proof (of Lemma 6). All parts by straightforward induction on the steps of proof outlines. □

Proof (of Lemma 7). The cases 2a and 2b are satisfied by the definition of the projection operator. Inductivity for the cases 2c and 2d are easy to show using Lemma 6 and the cases 2a and 2b of this lemma. If the order of the elements is unimportant, in the following we also use set notation for the values of the wait and notified variables. Correctness of the projection operation uses the results of this lemma and is formulated in Lemma 5. For the other cases we proceed by induction on the length of the run $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \acute{T}, \acute{\sigma} \rangle$ of the proof outline $prog'$.

In the base case of an initial configuration $\langle T_0, \sigma_0 \rangle$ (cf. page 11), the set T_0 contains exactly one thread (α, τ, stm), executing the non-synchronized main-statement of the program, i.e., $\neg owns(T_0 \downarrow prog, \alpha)$, and initially the lock of the only object α is set to *free*. Furthermore, the instance variables wait and notified of the initial object are set to $\emptyset$, and the *wait* and *notified* sets of the semantics are also empty.

For the inductive step, assume $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \grave{T}, \grave{\sigma} \rangle \longrightarrow \langle \acute{T}, \acute{\sigma} \rangle$. We distinguish on the kind of the last computation step.

Case: CALL_{start}, $\text{CALL}^{skip}_{start}$, RETURN_{run}, TRY, FINALLY, YRT, THROW_1, THROW_2, THROW_3, THROW_5
In these cases none of the concerned variables or predicates are touched, and the property follows directly by induction.

Case: ASS_{inst}, ASS_{loc}
Note that this case handles assignments, but not the observations of communication, object creation, and exception handling. Remember furthermore that the signaling mechanism is implemented in proof outlines by auxiliary assignments, and thus this case covers also the rules SIGNAL, SIGNAL_{skip}, and SIGNALALL.

If the assignment is not in a notify- or in a notifyAll-method representing notification, then the case is analogous to the above one.

Assume first that the assignment in the last computation step represents notification in a notify-method of the proof outline. If the wait set $\grave{\sigma}(\alpha)(\mathsf{wait})$ is empty, then no notification takes place; the property follows directly by induction. Thus assume that the wait set is not empty. I.e., a thread $\xi_1 \in \grave{T}$ notifies another thread $\xi_2 = (\alpha_2, \tau, stm) \circ \xi_2' \in \grave{T}$ in the *wait* set of α. Remember that notification is represented by a single assignment of the notifier, and thus the stack of the notified thread ξ_2 does not change. However, according to the projection definition, as the notifier changes the value of wait of α, the projection $\xi_2 \downarrow prog$ represents a thread being in the wait set in $\langle \grave{T}, \grave{\sigma} \rangle$ and being in the notified set in $\langle \acute{T}, \acute{\sigma} \rangle$.

The only relevant effect of the step is moving $(\alpha_2, n) \in \grave{\sigma}(\alpha)(\mathsf{wait})$ from the wait set into the notified set of α, where n is by induction the number of synchronized invocations of ξ_2 in α. Thus the properties 1a, 1b and 2e are automatically invariant. Induction implies also uniqueness of the representation of the wait and notified sets, i.e., α_2 is contained neither in $\grave{\sigma}(\alpha)(\mathsf{notified})$ nor in $\acute{\sigma}(\alpha)(\mathsf{wait})$. Thus moving the thread of α_2 from the wait into the notified set does not violate uniqueness of the representation.

The case for the assignment in the notifyAll-method is analogous, with the difference that all threads in the wait set get notified by ξ_1. The notifier sets the value of the auxiliary instance variable notified of α to $\grave{\sigma}(\alpha)(\mathsf{notified}) \dot{\cup} \grave{\sigma}(\alpha)(\mathsf{wait})$, whereas the corresponding wait variable gets the value $\emptyset$. By induction we have $\grave{\sigma}(\alpha)(\mathsf{notified}) \cap \grave{\sigma}(\alpha)(\mathsf{wait}) = \emptyset$, and thus the required properties are invariant under notification.

Case: NEW
Assume that the last step creates a new object, and executes the corresponding observation. Let $\alpha \in dom(\acute{\sigma})$. Then α either references the newly created object, or $\alpha \in dom(\grave{\sigma})$. In the first case $\alpha \notin dom(\grave{\sigma})$, and by the definition of global configurations (cf. page 10) there is no local configuration $(\alpha, \tau, stm) \in \grave{T}$, and the wait and notified set of α in $\grave{T}$ are empty. Since the last step doesn't add any local configurations to $\grave{T}$, we have $\alpha \neq \beta$ for all $(\beta, \tau, stm) \in \acute{T}$ and thus $\neg owns(\acute{T} \downarrow prog, \alpha)$. Since the lock of the new object is initialized to *free*, and wait and notified of α get the value $\emptyset$, the required property holds for the new object. In the second case, if $\alpha \in dom(\grave{\sigma})$, the property follows directly by induction.

Case: CALL
Let $\alpha \in dom(\acute{\sigma})$. Then also $\alpha \in dom(\grave{\sigma})$. If α is not the callee object, then the property holds directly by induction. If α is the callee object, the only new local configuration (α, τ, stm) in $\acute{T}$ represents the execution of the invoked method.

If the invoked method is non-synchronized, then the property follows by induction (invocations of monitor methods are covered by the $\text{CALL}_{monitor}$ case below). In the case of a synchronized method, let $\xi \in \grave{T}$ be the executing thread. The antecedent $\neg owns(\grave{T}\setminus\{\xi\} \downarrow prog, \alpha)$ implies by induction that, if there is no local configuration in the thread's stack executing a synchronized method of α then $\grave{\sigma}(\alpha)(\mathsf{lock}) = \mathit{free}$, and $\grave{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n)$ otherwise, where $(\alpha_0, \tau_0, stm_0)$ is the deepest configuration in the thread's stack and n the number of synchronized method invocations in the stack ξ. If in the state prior to the method invocation $\grave{\sigma}(\alpha)(\mathsf{lock}) = \mathit{free}$, then (α, τ, stm) is the only local configuration in $\acute{T}$ representing the execution of a synchronized method of α by a thread not in the wait or notified sets of α. Furthermore, the callee observation sets $\acute{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, 1)$, and thus the required property holds. In the second case, using the fact that the callee configuration is on top of its stack, the callee observation changes $\grave{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n)$ to $\acute{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n+1)$, and we get the property by Lemma 6 and by induction.

Case: $\text{CALL}_{monitor}$
Similarly to the case CALL, for $\alpha \in dom(\acute{\sigma})$ also $\alpha \in dom(\grave{\sigma})$, and if α is not the callee object, then the property holds by induction. In the case of the non-synchronized notify- and notifyAll-methods, none of the concerned variables or predicates are touched, and thus the property holds by induction again. So let $\xi \in \grave{T}$ be the executing thread invoking the non-synchronized wait-method of α.

The antecedent $owns(\xi \downarrow prog, \alpha)$ implies by induction $\grave{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n)$, where $(\alpha_0, \tau_0, stm_0)$ is the deepest configuration in the stack ξ and n is the number of its synchronized method invocations in α. Furthermore, since ξ does not yet execute a wait-method prior to the call, from $\xi \notin wait(\grave{T} \downarrow prog, \alpha) \cup notified(\grave{T} \downarrow prog, \alpha)$ we conclude by induction that α_0 is contained neither in wait nor in notified of α in $\grave{\sigma}$.

The execution places the thread into α's wait set and, since at most one thread can own a lock at a time, it gives the lock of α free, i.e., we have $\neg owns(\acute{T} \downarrow prog, \alpha)$. The corresponding callee observation extends $\grave{\sigma}(\alpha)(\mathsf{wait})$ with (α_0, n), and sets the lock-value of α to *free*. Thus the case follows by induction.

Case: RETURN
Assume $\alpha \in dom(\acute{\sigma}) = dom(\grave{\sigma})$. If α is not the callee object, or if the invoked method is non-synchronized, then the property holds directly by induction. Note that returning from the wait-method is covered by the RETURN_{wait} case below. So let $\xi \in \grave{T}$ be the thread of α_0 returning from a synchronized method of α; we denote the thread after execution by $\xi' \in \acute{T}$.

Since ξ is neither in the wait nor in the notified set of α, we get by definition $owns(\xi \downarrow prog, \alpha)$ prior to execution. If the given method is the only synchronized method of α executed by ξ, then in the successor configuration $\neg owns(\xi' \downarrow prog, \alpha)$, and from the invariant property that at most one thread can own a lock at a time we imply $\neg owns(\acute{T} \downarrow prog, \alpha)$. Otherwise, if ξ has reentrant synchronized method invocations in α, then the thread doesn't give the lock free upon return, i.e., in the successor state we still have $owns(\xi' \downarrow prog, \alpha)$.

Using $owns(\xi \downarrow prog, \alpha)$, we get by induction $\grave{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n)$, where n is the number of invocations of synchronized methods of α by ξ. The auxiliary variable lock of α is set by the callee augmentation to *free*, if $n = 1$, and to $(\alpha_0, n-1)$, otherwise. Since the auxiliary variables wait and notified are not touched, the property follows by induction.

Case: RETURN_{wait}
Assume that the thread $\xi \in \grave{T}$ of an object α_0 is returning from the wait-method of $\alpha \in dom(\acute{\sigma}) = dom(\grave{\sigma})$; we denote the thread after execution by $\xi' \in \acute{T}$.

The semantics assures $\neg owns(\grave{T} \downarrow prog, \alpha)$ and by definition $\xi \in notified(\grave{T} \downarrow prog, \alpha)$. We get by induction $\grave{\sigma}(\alpha)(\mathsf{lock}) = free$ and $(\alpha_0, n) \in \grave{\sigma}(\alpha)(\mathsf{notified})$, where n is the number of invocations of synchronized methods of α by ξ. After returning, the thread gets removed from the *notified*-set of α and gathers the lock of α, i.e., $\xi' \notin notified(\acute{T} \downarrow prog, \alpha)$ and $owns(\xi' \downarrow prog, \alpha)$.

The augmentation of the wait-method removes (α_0, n) from $\grave{\sigma}(\alpha)(\mathsf{notified})$; from the uniqueness of the representation follows $\alpha_0 \neq \beta$ for all $(\beta, m) \in \acute{\sigma}(\alpha)(\mathsf{notified})$. Furthermore, the observation sets the lock of α to (α_0, n), by which we get the required property.

Case: THROW_4
This case is analogous to the case RETURN. Remember that the observations of throw statements outside try-catch-finally blocks in synchronized methods decrement the lock value. □

Proof (of Lemma 8). Straightforward by the definition of augmentation. □

B.2 Proof of the soundness theorem

Proof (of the soundness Theorem 1). We prove the theorem by induction on the length of the computation, simultaneously for all parts of Definition 19.

For the initial case assume $dom(\sigma_0) = \{\alpha\}$, $\sigma_0(\alpha) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \alpha]$, $\tau_0 = \tau_{init}[\mathsf{thread} \mapsto \alpha]$, and let $\{p_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2 \rangle^{?call}\{p_3\}\, stm$ be the main statement. Then the initial configuration $\langle T_0', \sigma_0' \rangle$ of the proof outline satisfies the following: $\sigma_0' = \sigma_0[\alpha.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma_0(\alpha),\tau_0}]$, and for the stack we have $T_0' = \{(\alpha, \tau_0', stm)\}$ with $\tau_0' = \tau_0[\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma_0(\alpha),\tau_0}]$.

Let ω be a logical environment referring only to values existing in σ_0. As in σ_0 there exists exactly one object α being in its initial instance state, we have

$$\omega[z \mapsto \alpha], \sigma_0 \models_{\mathcal{G}} \mathsf{InitState}(z) \wedge \forall z'.\ z' = \mathsf{null} \vee z = z'\ ,$$

where z is of the type of the main class, and z' is a logical variable of type Object. Using the initial correctness condition we get

$$\omega[z \mapsto \alpha], \sigma_0 \models_{\mathcal{G}} (GI \wedge P_3(z) \wedge I(z)) \circ f_{obs} \circ f_{init}$$

with I the class invariant of α, $\vec{v}$ the local variables of the run-method of the main class, and

$$f_{init} = [\mathsf{this}, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]\ , \text{ and}$$
$$f_{obs} = [\vec{E}_2(z)/z.\vec{y}_2]\,.$$

Applying Lemma 4, we get for the global invariant $\omega', \sigma_0' \models_{\mathcal{G}} GI$ for $\omega' = \omega[z \mapsto \alpha][\vec{v} \mapsto \tau_0'(\vec{v})]$. Since GI may not contain free logical variables, its value does not depend on the logical environment, and therefore $\omega, \sigma_0' \models_{\mathcal{G}} GI$.

Similarly for the local property p_3, we get with Lemma 4 that $\omega', \sigma_0' \models_{\mathcal{L}} P_3(z)$. With Lemma 1 we get $\omega', \sigma_0'(\alpha), \tau_0' \models_{\mathcal{L}} pre(stm)$. Since $pre(stm)$ does not contain free logical variables, we get finally $\omega, \sigma_0'(\alpha), \tau_0' \models_{\mathcal{L}} pre(stm)$. Part 3 is analogous.

For the inductive step, assume $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle \grave{T}, \grave{\sigma}\rangle \longrightarrow \langle \acute{T}, \acute{\sigma}\rangle$ such that $\langle \grave{T}, \grave{\sigma}\rangle$ satisfies the conditions of Definition 19. Let ω be a logical environment referring only to values existing in $\acute{\sigma}$. We distinguish on the kind of the computation step $\langle \grave{T}, \grave{\sigma}\rangle \longrightarrow \langle \acute{T}, \acute{\sigma}\rangle$.

If the computation step is executed by a single local configuration, we use the local correctness conditions for inductivity of the executing local configuration's properties, and the interference freedom test for all other local configurations and the class invariants in $\langle \acute{T}, \acute{\sigma}\rangle$. For communication, invariance for the executing partners and the global invariant is shown using the cooperation test for communication. Communication itself does not affect the global state; invariance of the remaining properties under the corresponding observations is shown again with the help of the interference freedom test. The case for throwing exceptions outside try-blocks is similar. Finally for object creation, invariance for the global invariant, the creator local configuration, the created object's class invariant is assured by the conditions of the cooperation test for object creation; all other properties are shown to be invariant using the interference freedom test.

Case: $\textsc{Ass}_{inst}$, $\textsc{Ass}_{loc}$
Note that signaling is represented in proof outlines by auxiliary assignments, thus this case covers also the rules SIGNAL, SIGNALALL, and $\textsc{Signal}_{skip}$. Note furthermore that this case does not cover observations of communication, object creation, or exception throwing and handling.

Let the last computation step be the execution of an assignment in the local configuration $(\alpha, \grave{\tau}_1, \vec{y} := \vec{e}; stm_1) \in \grave{T}$ resulting in $(\alpha, \acute{\tau}_1, stm_1) \in \acute{T}$. According to the semantics, $\acute{\tau}_1 = \grave{\tau}_1[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$ and $\acute{\sigma} = \grave{\sigma}[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$.

Since assignments, that does not observe object creation, communication, or exception throwing, don't change the values of variables occurring in GI, part (2) is satisfied.

For part (1), assume $(\beta, \tau_2, stm_2) \in \acute{T}$. If $(\beta, \tau_2, stm_2) = (\alpha, \acute{\tau}_1, stm_1)$ is the executing local configuration, then by induction $\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} pre(\vec{y} := \vec{e})$. The local correctness condition implies that $\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} pre(stm_1)[\vec{e}/\vec{y}]$. Using the properties of the local substitution formulated in Lemma 3 we get $\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} pre(stm_1)$.

If otherwise (β, τ_2, stm_2) is not the executing local configuration, then it is contained in $\grave{T}$. If $\alpha \neq \beta$, i.e., the execution didn't take place in β, then $\grave{\sigma}(\beta) = \acute{\sigma}(\beta)$, and thus $\omega, \acute{\sigma}(\beta), \tau_2 \models_{\mathcal{L}} pre(stm_2)$ by induction. Otherwise let τ be $\grave{\tau}_1[\vec{v}' \mapsto \tau_2(\vec{v})]$, where $\vec{v} = dom(\tau_2)$ and $\vec{v}'$ fresh. Then Lemma 6, the induction assumptions, and the definition of $\mathsf{interleavable}$ imply

$$\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre(\vec{y} := \vec{e}) \wedge pre'(stm_2) \wedge \mathsf{interleavable}(pre(stm_2), \vec{y} := \vec{e}) \ ,$$

and with the interference freedom test we get $\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre'(stm_2)[\vec{e}/\vec{y}]$. Using the substitution Lemma 3 and the fact that, due to the renaming mechanism, no variables in $\vec{v}'$ may occur in $\vec{y}$, yields $\omega, \acute{\sigma}(\alpha), \tau_2 \models_{\mathcal{L}} pre(stm_2)$.

Part (3) is similar, using the fact that the class invariant may contain instance variables only, and thus its evaluation doesn't depend on the local state.

Case: CALL

Let $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} stm_1) \in \grave{T}$ be the caller configuration prior to method invocation, and let $(\alpha, \acute{\tau}_1, stm_1') \in \acute{T}$ and $(\beta, \acute{\tau}_2, stm_2) \in \acute{T}$ be the local configurations of the caller and the callee after execution. Let furthermore $\langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} stm_2$ be the invoked method's body and $\vec{u}$ its formal parameters. Directly after communication the callee has the local state $\check{\tau}_2 = \tau_{init}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$; after the caller observation, the global state is $\check{\sigma} = \grave{\sigma}[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$ and the caller's local state is updated to $\acute{\tau}_1 = \grave{\tau}_1[\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$. Finally, the callee observation updates its local state to $\acute{\tau}_2 = \check{\tau}_2[\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\check{\sigma}(\beta), \check{\tau}_2}]$ and the global state to $\acute{\sigma} = \check{\sigma}[\beta.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\check{\sigma}(\beta), \check{\tau}_2}]$. Let $\vec{v}_1$ denote $dom(\grave{\tau}_1)$ and assume $\grave{\omega} = \omega[z \mapsto \alpha][z' \mapsto \beta][\vec{v}_1 \mapsto \grave{\tau}_1(\vec{v}_1)]$.

The semantics assures $\alpha \neq null$ and $\beta = [\![e_0]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1} \neq null$, and we get with Lemma 1 and the definition of $\grave{\omega}$ that $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z \neq \mathsf{null} \wedge z' \neq \mathsf{null} \wedge E_0(z) = z'$.

If the method is synchronized and ξ is the stack of the executing thread in $\grave{T}$, then according to the transition rule $\neg owns(\grave{T} \setminus \{\xi\} \downarrow prog, \beta)$. Using Lemma 7 and Lemma 6 we get $\grave{\sigma}(\beta)(\mathsf{lock}) = \mathit{free} \vee \mathit{thread}(\grave{\sigma}(\beta)(\mathsf{lock})) = \grave{\tau}_1(\mathsf{thread})$ and thus $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$.

In the following let $p_1 = pre(u_{ret} := e_0.m(\vec{e}))$, $p_2 = pre(\vec{y}_1 := \vec{e}_1)$, $p_3 = post(\vec{y}_1 := \vec{e}_1)$, $q_1 = I_q$, $q_2 = pre(\vec{y}_2 := \vec{e}_2)$, and $q_3 = post(\vec{y}_2 := \vec{e}_2)$, where I_q is the class invariant of the callee. Then we have by induction $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} GI$, for the class invariant $\grave{\omega}, \grave{\sigma}(\beta), \grave{\tau}_1 \models_{\mathcal{L}} I_q$, and for the precondition of the call $\grave{\omega}, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_1$. Using the lifting lemma, the cooperation test for communication implies

$$\begin{aligned} \grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \ & (P_2(z) \wedge Q_2'(z'))[\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}'] \wedge \\ & (GI \wedge P_3(z) \wedge Q_3'(z'))[E_2'(z')/z'.\vec{y}_2'][E_1(z)/z.\vec{y}_1][\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}'] \ , \end{aligned}$$

where $\vec{v}$ contains the local variables of the callee without the formal parameters $\vec{u}$. Using the lifting lemma again but in the reverse direction and Lemma 4 results $\omega, \acute{\sigma} \models_{\mathcal{G}} GI$, and thus part (2). Note that in the annotation no free logical variables occur, and thus the values of assertions in a proof outline do not depend on the logical environment. Furthermore, using the same lemmas we get

$$\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_2 \qquad \omega, \grave{\sigma}(\beta), \grave{\tau}_2 \models_{\mathcal{L}} q_2$$
$$\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_3 \qquad \omega, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} q_3 \ .$$

Thus part (1) is satisfied for the local configurations involved in the last computation step. All other configurations (γ, τ_3, stm_3) in $\acute{T}$ are also in $\grave{T}$. If $\gamma \neq \alpha$ and $\gamma \neq \beta$, then $\grave{\sigma}(\gamma) = \acute{\sigma}(\gamma)$, and thus $\omega, \acute{\sigma}(\gamma), \tau_3 \models_{\mathcal{L}} pre(stm_3)$ by induction.

Assume next $\gamma = \alpha$ and $\alpha \neq \beta$, and let τ be $\grave{\tau}_1[\vec{v}' \mapsto \tau_3(\vec{v})]$, where $\vec{v} = dom(\tau_3)$. Then Lemma 6, the induction assumptions, and the definition of the assertion $\mathsf{interleavable}$ imply with the interference freedom test $\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre'(stm_3)[\vec{e}_1/\vec{y}_1]$. The substitution Lemma 3 and the fact that, due to the renaming mechanism, no local variables in $\vec{v}'$ occur in $\vec{y}_1$, yield $\omega, \check{\sigma}(\alpha), \tau_3 \models_{\mathcal{L}} pre(stm_3)$. Now, since $\beta \neq \alpha$, the callee observation also does not change the caller's instance state, and we have $\check{\sigma}(\alpha) = \acute{\sigma}(\alpha)$. Thus we get $\omega, \acute{\sigma}(\alpha), \tau_3 \models_{\mathcal{L}} pre(stm_3)$.

The case $\gamma = \beta$ and $\alpha \neq \beta$ is similar. Communication and caller observation do not change the instance state of β, i.e., $\grave{\sigma}(\beta) = \check{\sigma}(\beta)$. The interference freedom test results $\omega, \check{\sigma}(\beta), \tau \models_{\mathcal{L}} pre'(stm_3)[\vec{e}_2/\vec{y}_2]$ with $\tau = \check{\tau}_2[\vec{v}' \mapsto \tau_3(\vec{v})]$. Due to the renaming mechanism, we conclude with the local substitution lemma that $\omega, \acute{\sigma}(\beta), \acute{\tau} \models_{\mathcal{L}} pre'(stm_3)$ with $\acute{\tau}(\vec{v}') = \tau_3(\vec{v})$, and thus $\omega, \acute{\sigma}(\beta), \tau_3 \models_{\mathcal{L}} pre(stm_3)$.

For the last case $\gamma = \alpha = \beta$ note that, according to the restrictions on the augmentation, the caller may not change the instance state. Thus the same arguments as for $\gamma = \beta$ and $\alpha \neq \beta$ apply. I.e., part (1) is satisfied.

Part (3) is analogous: The interference freedom test implies $\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} I_p$, where I_p is the class invariant of the caller. Since I_p may contain instance variables only, its evaluation doesn't depend on the local state. Similarly for the callee, $\omega, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} I_q$. The state of other objects is not changed in the last computation step, and we get the required property.

Case: $\textsc{Call}_{start}$, $\textsc{Call}^{skip}_{start}$

These cases are analogous to the above one, where we additionally need $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \neg z'.\mathsf{started}$ and $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z'.\mathsf{started}$, respectively, to be able to apply the cooperation test. The above properties result from the transition antecedents $\neg started(\grave{T}, \beta)$ and $started(\grave{T}, \beta)$, respectively, using Lemma 8 and $\grave{\omega}(z') = \beta$.

Case: $\textsc{Call}_{monitor}$

As above, where $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ is implied by the transition antecedent $owns(\xi \downarrow prog, \beta)$ for the executing thread ξ, and Lemma 6.

Case: $\textsc{Return}$

This case is analogous to the $\textsc{Call}$ case, where we define q_1 as the precondition of the corresponding return statement instead of the callee class invariant. The requirement $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$ of the cooperation test results from the fact that the values of formal parameters may not change during method execution, and that the method invocation statements may not contain instance variables, so that the values of the formal parameters and the expressions in the method invocation statement are untouched during the execution of the invoked method.

For the application of the interference freedom test, to show the validity of the interleavable predicate, we use the fact that the assertion $pre(stm_3)$ neither describes the caller nor the callee, since the corresponding local configuration is not involved in the execution.

Case: $\textsc{Return}_{run}$
Similar to the return case.

Case: $\textsc{Return}_{wait}$
In this case the antecedent $\neg owns(\grave{T} \downarrow prog, \beta)$ of the transition rule together with Lemma 7 imply $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z'.\mathsf{lock} = \mathsf{free}$. Furthermore, the executing thread is in the notified set prior to execution, and the same lemma yields that the executing thread is registered in $\grave{\sigma}(\beta)(\mathsf{notified})$, i.e., $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \mathsf{thread}' \in z'.\mathsf{notified}$.

Case: $\textsc{Throw}_4$
This case is similar to the $\textsc{Return}$ case, where q_1 is the precondition of the given throw statement.

Case: $\textsc{Try}$
Let the last computation step be the entering of a try-catch-finally block with observation $\vec{y} := \vec{e}$, executed in the local configuration $(\alpha, \grave{\tau}_1, \grave{stm}_1) \in \grave{T}$, resulting in $(\alpha, \acute{\tau}_1, \acute{stm}_1) \in \acute{T}$. According to the semantics, directly after entering the block but before the corresponding observation we have $\check{\tau}_1 = \grave{\tau}_1[\mathsf{exc} \mapsto [\![\mathsf{exc}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1} \circ null]$ and $\check{\sigma} = \grave{\sigma}$. After executing the observation we get $\acute{\tau}_1 = \check{\tau}_1[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\check{\sigma}(\alpha), \check{\tau}_1}]$ and $\acute{\sigma} = \check{\sigma}[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\check{\sigma}(\alpha), \check{\tau}_1}]$.

Since observations of try keywords must not change the values of variables occurring in GI, part (2) is satisfied.

For part (1), assume $(\beta, \tau_2, stm_2) \in \acute{T}$. If $(\beta, \tau_2, stm_2) = (\alpha, \acute{\tau}_1, \acute{stm}_1)$ is the executing local configuration, then by induction $\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} pre(\grave{stm}_1)$. The local correctness condition implies $\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} pre(\acute{stm}_1)[\vec{e}/\vec{y}][\mathsf{exc} \circ \mathsf{null}/\mathsf{exc}]$. Using the properties of the local substitution formulated in Lemma 3 we get $\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} pre(\acute{stm}_1)$.

If otherwise (β, τ_2, stm_2) is not the executing local configuration, then it is contained in $\grave{T}$. If $\alpha \neq \beta$, i.e., the execution didn't take place in β, then $\grave{\sigma}(\beta) = \acute{\sigma}(\beta)$, and thus $\omega, \acute{\sigma}(\beta), \tau_2 \models_{\mathcal{L}} pre(stm_2)$ by induction. Otherwise, analogously to the argumentation above, the local correctness Condition 10 implies $\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} pre(\vec{y} := \vec{e})[\mathsf{exc} \circ \mathsf{null}/\mathsf{exc}]$. Using the properties of the local substitution formulated in Lemma 3 we get $\omega, \check{\sigma}(\alpha), \check{\tau}_1 \models_{\mathcal{L}} pre(\vec{y} := \vec{e})$.

Let τ be $\check{\tau}_1[\vec{v}' \mapsto \tau_2(\vec{v})]$, where $\vec{v} = dom(\tau_2)$ and $\vec{v}'$ fresh. Then Lemma 6, the induction assumptions, and the definition of interleavable imply

$$\omega, \check{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre(\vec{y} := \vec{e}) \wedge pre'(stm_2) \wedge \mathsf{interleavable}(pre(stm_2), \vec{y} := \vec{e}) \; ,$$

and with the interference freedom test we get $\omega, \check{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre'(stm_2)[\vec{e}/\vec{y}]$. Using the substitution Lemma 3 and the fact that, due to the renaming mechanism, no variables in $\vec{v}'$ may occur in $\vec{y}$, yields $\omega, \acute{\sigma}(\alpha), \tau_2 \models_{\mathcal{L}} pre(stm_2)$.

Part (3) is similar, using the fact that the class invariant may contain instance variables only, and thus its evaluation doesn't depend on the local state.

Case: FINALLY, YRT
These cases are analogous to the above one, where for FINALLY we have $\check{\tau}_1 = \grave{\tau}_1$, and for YRT $\check{\tau}_1 = \grave{\tau}_1[\mathsf{exc}, \mathsf{top} \mapsto [\![\mathsf{head}(\mathsf{exc})]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}_1}, [\![\mathsf{tail}(\mathsf{exc})]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}_1}]$; the substitution $[\mathsf{exc} \circ \mathsf{null}/\mathsf{exc}]$ is replaced accordingly.

Case: THROW$_1$
Let $(\alpha, \grave{\tau}, s\grave{t}m) \in \grave{T}$ with $s\grave{t}m = \mathsf{throw}\, e;\ \langle \vec{y} := \vec{e} \rangle^{throw}\ stm_0;\ \mathsf{catch}\,(c_1\, u_1)\, stm_1;\ \ldots;\ \mathsf{catch}\,(c_n\, u_n)\, stm_n;\ \mathsf{finally}\, stm_{n+1}\ \mathsf{yrt};\, stm_{n+2}$ be the executing local configuration prior to the computation step, resulting in $(\alpha, \acute{\tau}, s\acute{t}m) \in \acute{T}$ with $s\acute{t}m = stm_i; \mathsf{finally}\, stm_{n+1}\, \mathsf{yrt};\, stm_{n+2}$ after execution. According to the semantics, $[\![e]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}} \in Val^{c_i}$ for some $1 \le i \le n$, implying $[\![e{\neq}\mathsf{null} \wedge \mathsf{hastype}(e, c_i)]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}}$. Furthermore, from $\forall 1 \le j < i.\, [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\grave{\tau}} \notin Val^{c_j}$ we conclude $[\![\forall 1 \le j < i. \neg\, \mathsf{hastype}(e, c_j)]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}}$.

Directly after exception throwing we have $\check{\tau} = \grave{\tau}[u_i \mapsto [\![e]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}}]$ and $\check{\sigma} = \grave{\sigma}$. The observation modifies the states resulting in $\acute{\tau} = \check{\tau}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\check{\tau}}]$ and $\acute{\sigma} = \grave{\sigma}[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\check{\tau}}]$.

Since observations of exception throwing inside try-catch-finally blocks must not change the values of variables occurring in GI, part (2) is satisfied.

For part (1), assume $(\beta, \tau', stm') \in \acute{T}$. If $(\beta, \tau', stm') = (\alpha, \acute{\tau}, s\acute{t}m)$ is the executing local configuration, then by induction $\omega, \grave{\sigma}(\alpha), \grave{\tau} \models_{\mathcal{L}} pre(s\grave{t}m)$. The local correctness Condition 17 implies $\omega, \grave{\sigma}(\alpha), \grave{\tau} \models_{\mathcal{L}} pre(s\acute{t}m)[\vec{e}/\vec{y}][e/u_i]$. Using the properties of the local substitution formulated in Lemma 3 we get $\omega, \acute{\sigma}(\alpha), \acute{\tau} \models_{\mathcal{L}} pre(s\acute{t}m)$.

If otherwise (β, τ', stm') is not the executing local configuration, then it is contained in $\grave{T}$. If $\alpha \neq \beta$, i.e., the execution didn't take place in β, then $\grave{\sigma}(\beta) = \acute{\sigma}(\beta)$, and thus $\omega, \acute{\sigma}(\beta), \tau' \models_{\mathcal{L}} pre(stm')$ by induction. Otherwise, the induction assumptions, the local correctness Condition 16, and the local substitution Lemma 3 imply $\omega, \grave{\sigma}(\alpha), \check{\tau}_1 \models_{\mathcal{L}} pre(\vec{y} := \vec{e})$.

Let τ be $\check{\tau}[\vec{v}' \mapsto \tau'(\vec{v})]$, where $\vec{v} = dom(\tau')$ and $\vec{v}'$ fresh. Then Lemma 6, the induction assumptions, and the definition of interleavable imply

$$\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre(\vec{y} := \vec{e}) \wedge pre'(stm') \wedge \mathsf{interleavable}(pre(stm'), \vec{y} := \vec{e}) ,$$

and with the interference freedom test we get $\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre'(stm')[\vec{e}/\vec{y}]$. Using the substitution Lemma 3 and the fact that, due to the renaming mechanism, no variables in $\vec{v}'$ may occur in $\vec{y}$, yields $\omega, \acute{\sigma}(\alpha), \tau' \models_{\mathcal{L}} pre(stm')$.

Part (3) is similar, using the fact that the class invariant may contain instance variables only, and thus its evaluation doesn't depend on the local state.

Case: THROW$_2$, THROW$_3$, THROW$_5$
These cases are similar to the above one. None of these statements may change the values of variables occurring in the global invariant, and thus part (2) is satisfied.

The induction assumptions and the semantics assures that the antecedents of the corresponding local conditions hold in the configuration prior to execution. Satisfaction of the local conditions and the local substitution lemma imply that the precondition of the statement of the executing local configuration hold after the computation step.

For the other local configurations, local correctness assures additionally, that the precondition of the attached observation hold directly before its execution. Again, we use induction assumption, satisfaction of the interference freedom conditions, and the local substitution lemma to show that the given assertion attached to the control point of the non-executing local configuration hold after observation.

Case: NEW

Let $(\alpha, \grave{\tau}_1, u := \mathsf{new}; \langle \vec{y} := \vec{e} \rangle^{new}\, stm_1) \in \grave{T}$ be the local configuration of the executing thread prior to object creation, and $(\alpha, \acute{\tau}_1, stm_1) \in \acute{T}$ after it. Object creation updates the global state to $\check{\sigma} = \grave{\sigma}[\beta \mapsto \sigma^{init}_{inst}[\mathsf{this} \mapsto \beta]]$, where $\beta \notin dom(\grave{\sigma})$; the executing thread's local state gets updated to $\check{\tau}_1 = \grave{\tau}_1[u \mapsto \beta]$. After observation we have $\acute{\tau}_1 = \check{\tau}_1[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\check{\sigma}(\alpha), \check{\tau}_1}]$ and for the global state $\acute{\sigma} = \check{\sigma}[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\check{\sigma}(\alpha), \check{\tau}_1}]$.

In the following let $p_1 = pre(u := \mathsf{new})$, $p_2 = pre(\vec{y} := \vec{e})$, and $p_3 = post(\vec{y} := \vec{e})$. By induction $\omega, \grave{\sigma} \models_{\mathcal{G}} GI$ and $\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_1$. Using the lifting lemma we get $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} GI \wedge P_1(z)$ for $\grave{\omega} = \omega[z \mapsto \alpha][\vec{v}_1 \mapsto \grave{\tau}_1(\vec{v}_1)]$ and $\vec{v}_1$ the variables from the domain of $\grave{\tau}_1$. Lemma 2 yields $\grave{\omega}[z' \mapsto dom(\grave{\sigma})][u \mapsto \beta], \check{\sigma} \models_{\mathcal{G}} (GI \wedge (\exists u.\ P_1(z))) \downarrow z'$. Note that GI may not contain free logical variables, and thus its evaluation does not depend on the logical environment. The newly created object with a fresh identity is in its initial instance state, implying $\grave{\omega}[z' \mapsto dom(\grave{\sigma})][u \mapsto \beta], \check{\sigma} \models_{\mathcal{G}} \mathsf{Fresh}(z', u)$. Thus the cooperation test for object creation implies

$$\grave{\omega}[u \mapsto \beta], \check{\sigma} \models_{\mathcal{G}} P_2(z) \wedge I_{\mathsf{new}}(u) \wedge (GI \wedge P_3(z))[\vec{E}(z)/z.\vec{y}]\ ,$$

where I_{new} is the class invariant of the new object. Using the lifting lemma again but in the reverse direction and Lemma 4 results $\omega, \acute{\sigma} \models_{\mathcal{G}} GI$, and thus part (2). Note that in the annotation no free logical variables occur, and thus the values of assertions do not depend on the logical environment.

Furthermore, using the substitution lemmas we get

$$\omega, \check{\sigma}(\alpha), \check{\tau}_1 \models_{\mathcal{L}} p_2\ , \qquad \omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_3\ , \quad \text{and} \quad \omega, \acute{\sigma}(\beta), \tau \models_{\mathcal{L}} I_{\mathsf{new}}$$

for all τ. For the class invariant of the executing thread, the interference freedom test implies $\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} I$, where I is the class invariant of α. Since I may contain instance variables only, its evaluation doesn't depend on the local state, and the required property holds. The state of other objects not involved in the last step is not changed in the last computation step, and part (3) is satisfied.

Furthermore, part (1) is satisfied for the local configuration involved in the last computation step. All other configurations (γ, τ_2, stm_2) in $\acute{T}$ are also in $\grave{T}$ and $\gamma \neq \beta$. If $\gamma \neq \alpha$, then $\grave{\sigma}(\gamma) = \acute{\sigma}(\gamma)$, and thus $\omega, \acute{\sigma}(\gamma), \tau_2 \models_{\mathcal{L}} pre(stm_2)$ by induction.

Assume now $\gamma = \alpha$, and let τ be $\check{\tau}_1[\vec{v}' \mapsto \tau_2(\vec{v})]$, where $\vec{v} = dom(\tau_2)$. Then, since $\grave{\sigma}(\alpha) = \check{\sigma}(\alpha)$, Lemma 6, the induction assumptions, and the definition of interleavable imply using the interference freedom test that $\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre'(stm_2)[\vec{e}/\vec{y}]$. The substitution Lemma 3 and the fact that, due to the renaming mechanism, no local variables in $\vec{v}'$ occur in $\vec{y}$, yields $\omega, \acute{\sigma}(\alpha), \tau_2 \models_{\mathcal{L}} pre(stm_2)$. I.e., part (1) is satisfied. □

Proof (of the soundness Corollary 1). The proof is straightforward using the soundness Lemma 1. □

C Completeness proof

The following lemma states that the variable loc indeed stores the current control point of a thread:

Lemma 12. *Let $\langle T, \sigma\rangle$ be a reachable configuration of $prog'$ and $(\alpha, \tau, stm) \in T$. Then $\tau(\mathsf{loc}) \equiv stm$.*

Proof (of Lemma 12). Straightforward by the definition of augmentation. □

Proof (of the local merging Lemma 9). Assume two computations $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle \acute{T}_1, \acute{\sigma}_1\rangle$ and $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle \acute{T}_2, \acute{\sigma}_2\rangle$ of $prog'$, and let $(\alpha, \tau, stm) \in \acute{T}_1$ with $\alpha \in dom(\acute{\sigma}_1) \cap dom(\acute{\sigma}_2)$ and $\acute{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{inst})$. We prove $(\alpha, \tau, stm) \in \acute{T}_2$ by induction over the sum of the length of the computations.

In the initial case both $\acute{T}_1$ and $\acute{T}_2$ contain the same single initial local configuration, and thus the property holds.

For the inductive case, let $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ be the last steps of the computations. The augmentation definition implies that each computation step appends at most one element to the instance history of α. If $\grave{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \acute{\sigma}_1(\alpha)(\mathsf{h}_{inst})$, then, by the definition of the augmentation, $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ did not execute in α, i.e., $(\alpha, \tau, stm) \in \grave{T}_1$, and the property follows by induction. The case for $\grave{\sigma}_2(\alpha)(\mathsf{h}_{inst}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{inst})$ is analogous. Thus assume in the following $\acute{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \grave{\sigma}_1(\alpha)(\mathsf{h}_{inst}) \circ (\sigma^1_{inst}, \tau_1)$ and $\acute{\sigma}_2(\alpha)(\mathsf{h}_{inst}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{inst}) \circ (\sigma^2_{inst}, \tau_2)$. From $\acute{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{inst})$ we conclude that $\grave{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{inst})$ and $(\sigma^1_{inst}, \tau_1) = (\sigma^2_{inst}, \tau_2)$.

Since $\acute{\sigma}_1(\alpha)(\mathsf{h}_{inst}) \neq \grave{\sigma}_1(\alpha)(\mathsf{h}_{inst})$, the computation step $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ executed some statements in α. If there is only one local configuration in α that was involved in the step, then the augmentation definition and the local substitution lemma imply that its resulting local configuration in $\acute{T}_1$ is given by (α, τ_1, stm_1) with $stm_1 \equiv \tau_1(\mathsf{loc})$. From $(\sigma^1_{inst}, \tau_1) = (\sigma^2_{inst}, \tau_2)$ we conclude that the same local configuration executed in $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$. Thus, either $(\alpha, \tau, stm) \in \acute{T}_1$ is the executing configuration (α, τ_1, stm_1) and then it is also in $\acute{T}_2$, or not, and then it is in $\grave{T}_1$, by induction in $\grave{T}_2$, and since it wasn't involved in the execution $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$, also in $\acute{T}_2$.

If otherwise there are two local configurations in α involved in $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$, then $(\sigma^1_{inst}, \tau_1)$ specifies the callee's instance local state. However, due to the built-in auxiliary variables, the identity of the caller local configuration is also stored in τ_1, in the formal parameter caller of the callee. The caller configuration is in $\grave{T}_1$, and by induction in $\grave{T}_2$. Furthermore, since there are no two local configurations with the same identity in a reachable configuration, both steps execute in the same instance local configuration.

Thus, either $(\alpha, \tau, stm) \in \acute{T}_1$ is one of the executing configurations and then it is also in $\acute{T}_2$, or not, and then it is in $\grave{T}_1$, by induction in $\grave{T}_2$, and since it wasn't involved in the execution, also in $\acute{T}_2$. □

Proof (of the global merging Lemma 10). Assume two reachable configurations $\langle \acute{T}_1, \acute{\sigma}_1 \rangle$ and $\langle \acute{T}_2, \acute{\sigma}_2 \rangle$ and let $\alpha \in dom(\acute{\sigma}_1) \cap dom(\acute{\sigma}_2)$ satisfying $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$. We show that there exists a reachable $\langle \acute{T}, \acute{\sigma} \rangle$ with $dom(\acute{\sigma}) = dom(\acute{\sigma}_2)$, $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$, and $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2) \setminus \{\alpha\}$. We proceed by induction on the sum of the lengths of the computations.

In the base case we are given $\langle \acute{T}_1, \acute{\sigma}_1 \rangle = \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ and the property trivially holds.

For the inductive step, let $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ be the last steps of the computations.

If $\alpha \notin dom(\grave{\sigma}_1)$ or $\alpha \notin dom(\grave{\sigma}_2)$, then α was created in one of the last steps, and thus $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm}) = \epsilon$. That means, no methods of α were involved yet, i.e., α is in its initial instance state $\acute{\sigma}_1(\alpha) = \acute{\sigma}_2(\alpha) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \alpha]$; in this case $\langle \acute{T}_2, \acute{\sigma}_2 \rangle$ already satisfies the requirements. Assume in the following $\alpha \in dom(\grave{\sigma}_1) \cap dom(\grave{\sigma}_2)$. We distinguish whether the last computation steps update the communication history of α or not.

Case: $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$
In this case $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$ doesn't execute any non-self communication or object creation in α. By induction there is a computation $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \grave{T}, \grave{\sigma} \rangle$ leading to a configuration such that $\grave{\sigma}(\alpha) = \grave{\sigma}_1(\alpha)$ and $\grave{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2) \setminus \{\alpha\}$.

In case $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$ does not execute in α at all, i.e., $\grave{\sigma}_1(\alpha) = \acute{\sigma}_1(\alpha)$, then $\langle \grave{T}, \grave{\sigma} \rangle$ already satisfies the requirements.

Otherwise, the local configurations in $\grave{T}_1$ which execute in α and which are involved in the computation step $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$ are by the local merging Lemma 9 also in $\grave{T}$. Furthermore, from $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$ we conclude that they don't execute any non-self communication or object creation, and thus their enabledness and effect depends only on the instance state of α. That means, the same computation as in $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$ can be executed in $\langle \grave{T}, \grave{\sigma} \rangle$, leading to a reachable global configuration satisfying the requirements.

Case: $\grave{\sigma}_2(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$
In this case $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ does not execute any non-self communication or object creation involving α. By induction, there is a reachable $\langle \grave{T}, \grave{\sigma} \rangle$ with $\grave{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$ and $\grave{\sigma}(\beta) = \grave{\sigma}_2(\beta)$ for all $\beta \in dom(\grave{\sigma}_2) \setminus \{\alpha\}$.

If $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ performs a step within α, then, according to the case assumption, it executes exclusively within α. This means, $\grave{\sigma}_2(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2)\backslash\{\alpha\}$, and $\langle \grave{T}, \grave{\sigma}\rangle$ already satisfies the required properties.

If otherwise $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ does not execute in α, then all local configurations in $\grave{T}_2$, executing in an object different from α, are also in $\grave{T}$; this follows from $\grave{\sigma}_2(\beta) = \grave{\sigma}(\beta)$ for all $\beta \in dom(\grave{\sigma}_2)\backslash\{\alpha\}$, and with the help of the local merging Lemma 9 applied to $\langle \grave{T}, \grave{\sigma}\rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2\rangle$. The enabledness of local configurations, whose execution does not involve α, are independent of the instance state of α; furthermore, the effect of their execution neither influences the instance state of α nor depends on it. Thus in $\langle \grave{T}, \grave{\sigma}\rangle$ we can execute the same computation steps as in $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$, leading to a reachable configuration with the required properties.

Case: $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) \neq \acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$ and $\grave{\sigma}_2(\alpha)(\mathsf{h}_{comm}) \neq \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$
In this case finally both $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ execute some object creation or non-self communication in α, including exception throwing between different objects. We show that in this case $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$ implies also $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{comm})$, and thus by induction there is a computation leading to a configuration $\langle \grave{T}, \grave{\sigma}\rangle$ such that $dom(\grave{\sigma}) = dom(\grave{\sigma}_2)$, $\grave{\sigma}(\alpha) = \grave{\sigma}_1(\alpha)$, and $\grave{\sigma}(\beta) = \grave{\sigma}_2(\beta)$ for all other objects $\beta \in dom(\grave{\sigma}_2)\backslash\{\alpha\}$.

Furthermore, combining those local configurations involved in $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ which execute within α with those in $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ which execute outside α, we can define a computation $\langle \grave{T}, \grave{\sigma}\rangle \longrightarrow \langle \acute{T}, \acute{\sigma}\rangle$ such that $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$ and $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ for all other objects $\beta \in dom(\acute{\sigma}_2)\backslash\{\alpha\}$.

The case assumptions imply, that the last elements of the communication histories $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$ and $\acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$ were appended in the last computation steps; $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$ imply that the last elements are equal.

According to the augmentation, each computation step extends the communication history of α with at most one element. Thus we get $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{comm})$, and by induction there is a reachable $\langle \grave{T}, \grave{\sigma}\rangle$ with $dom(\grave{\sigma}) = dom(\grave{\sigma}_2)$, $\grave{\sigma}(\alpha) = \grave{\sigma}_1(\alpha)$, and $\grave{\sigma}(\beta) = \grave{\sigma}_2(\beta)$ for all $\beta \in dom(\grave{\sigma}_2)\backslash\{\alpha\}$.

Note that the last elements of the communication histories $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$ and $\acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$ record the kind of execution, and so we know that both steps execute the same kind of communication in α. Furthermore, the last elements record also the identity of the local configuration executing in α, the communication partner of α, and the communicated values, which are consequently also equal.

We distinguish on the kind of the computation step $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$:

Subcase: NEW
In this case $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) \circ (\alpha, null, (\mathsf{new}^c\gamma, thread_\alpha))$, where $thread_\alpha$ is the identity of the creator thread as specified by its local variable thread, and γ is the newly created object.

From the preliminary observations we conclude that $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ creates the same new object γ being in the same initial state; furthermore, it leaves the states of all objects from $dom(\grave{\sigma}_2)\backslash\{\alpha\}$ untouched.

As $\grave{\sigma}(\alpha) = \grave{\sigma}_1(\alpha)$, the local merging Lemma 9 implies that the local configuration of the creator in $\grave{T}_1$ is also contained in $\grave{T}$. Thus, since $\gamma \notin dom(\grave{\sigma}_2) = dom(\grave{\sigma})$, the same computation step as in $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$ can be executed also in $\langle \grave{T}, \grave{\sigma} \rangle$, leading to a reachable configuration $\langle \acute{T}, \acute{\sigma} \rangle$ with $Val^{\mathsf{Object}}(\acute{\sigma}) = Val^{\mathsf{Object}}(\grave{\sigma}) \mathbin{\dot{\cup}} \{\gamma\} = Val^{\mathsf{Object}}(\grave{\sigma}_2) \mathbin{\dot{\cup}} \{\gamma\} = Val^{\mathsf{Object}}(\acute{\sigma}_2)$, $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$, and $\acute{\sigma}(\beta) = \grave{\sigma}(\beta) = \grave{\sigma}_2(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\grave{\sigma}_2) \backslash \{\alpha\}$. Finally, for the newly created object we have $\acute{\sigma}(\gamma) = \acute{\sigma}_2(\gamma) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \gamma]$, and thus $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2) \backslash \{\alpha\}$.

Subcase: CALL

Assume first that α is the caller object and $\beta \neq \alpha$ the callee. According to the preliminary observations, also $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ executes the invocation of the same method of β, where α is the caller and β the callee. Furthermore, by the local merging lemma, the caller local configuration from $\grave{T}_1$ is also in $\grave{T}$, and its execution is also enabled in $\langle \grave{T}, \grave{\sigma} \rangle$. The last property holds also for synchronized and monitor methods, since the invocation of the same method of β by the same thread is enabled in $\langle \grave{T}_2, \grave{\sigma}_2 \rangle$, and $\grave{\sigma}_2(\beta) = \grave{\sigma}(\beta)$.

Thus the caller local configuration from $\grave{T}_1$ can execute the method invocation in $\langle \grave{T}, \grave{\sigma} \rangle$, leading to a reachable configuration $\langle \acute{T}, \acute{\sigma} \rangle$ with $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$. Furthermore, $\langle \grave{T}, \grave{\sigma} \rangle \longrightarrow \langle \acute{T}, \acute{\sigma} \rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ execute the same callee observation in the same instance state $\grave{\sigma}_2(\beta) = \grave{\sigma}(\beta)$ and the same initial local state after the communication of the same actual parameter values, and thus $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$. The states of other objects are not touched, and thus $\langle \acute{T}, \acute{\sigma} \rangle$ satisfies the required properties.

Similarly, if the callee object is α, then the same caller local configuration as in $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ can execute in $\langle \grave{T}, \grave{\sigma} \rangle$ leading to a reachable configuration satisfying the requirements.

Subcase: RETURN, THROW$_4$

These cases are analogous to the above case for CALL. The computation $\langle \grave{T}, \grave{\sigma} \rangle \longrightarrow \langle \acute{T}, \acute{\sigma} \rangle$ is constructed from the execution of the local configuration in α which executes in $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$, together with the execution of the communication partner of α which executes in $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$. □

Lemma 13 (Initial correctness). *The proof outline prog′ satisfies the initial conditions of Definition 12.*

Proof (of Lemma 13). Let $\{p_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2 \rangle^{?call}\{p_3\}\ stm;\mathsf{return}$ be the main statement with local variables $\vec{v}$, and let I be the class invariant of the main class. We have to show for arbitrary $\sigma \in \Sigma$ and $\omega \in \Omega$ referring only to values existing in σ, that

$$\begin{aligned}\omega, \sigma \models_{\mathcal{G}} \mathsf{InitState}(z) \wedge (\forall z'.\ z' = \mathsf{null} \vee z = z') \rightarrow \\ P_2(z) \circ f_{init} \wedge (GI \wedge P_3(z) \wedge I(z)) \circ f_{obs} \circ f_{init}\ ,\end{aligned}$$

where z is of the type of the main class, z' of type Object, and where $f_{init} = [z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]$ and $f_{obs} = [\vec{E}_2(z)/z.\vec{y}_2]$. We observe that

$$\omega, \sigma \models_{\mathcal{G}} \mathsf{InitState}(z) \wedge (\forall z'.\ z' = \mathsf{null} \vee z' = z)$$

implies that σ is the initial global state prior to the execution of the callee observation at the beginning of the main statement, i.e., defining exactly one existing object $\omega(z) = \alpha$ being in its initial instance state $\sigma(\alpha) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \alpha]$. We start transforming the right-hand side using the substitution Lemmas 4 and 1:

$$
\begin{aligned}
& [\![P_2(z)[z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]]\!]_{\mathcal{G}}^{\omega,\sigma} \\
= {} & [\![P_2(z)[z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}]]\!]_{\mathcal{G}}^{\omega[\vec{v} \mapsto \mathsf{Init}(\vec{v})],\sigma} \\
= {} & [\![P_2(z)]\!]_{\mathcal{G}}^{\omega[\vec{v} \mapsto \mathsf{Init}(\vec{v})][\mathsf{thread} \mapsto \alpha],\sigma} \\
= {} & [\![p_2]\!]_{\mathcal{L}}^{\omega,\sigma(\alpha),\tau}
\end{aligned}
$$

with τ defined by $\tau_{init}[\mathsf{thread} \mapsto \alpha][\mathsf{caller} \mapsto (null, 0, null)]$. The above value is *true*, since the run-method of the main class is initially invoked in the given context.

For the global invariant we get similarly

$$
\begin{aligned}
& [\![GI[\vec{E}_2(z)/z.\vec{y}_2][z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]]\!]_{\mathcal{G}}^{\omega,\sigma} \\
= {} & [\![GI[\vec{E}_2(z)/z.\vec{y}_2]]\!]_{\mathcal{G}}^{\omega[\vec{v} \mapsto \mathsf{Init}(\vec{v})][\mathsf{thread} \mapsto \alpha],\sigma} \\
= {} & [\![GI]\!]_{\mathcal{G}}^{\omega',\sigma'} \\
= {} & [\![GI]\!]_{\mathcal{G}}^{\omega,\sigma'}
\end{aligned}
$$

for some logical environment ω' and σ' given by $\sigma[\alpha.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}]$. In the last step we used the restriction that the global invariant may not contain free logical variables. The step before made use of the following equation for $\vec{E}_2(z)$, which we get using Lemma 1 and with the fact that $\vec{e}_2$ does not contain logical variables:

$$
\begin{aligned}
[\![\vec{E}_2(z)]\!]_{\mathcal{G}}^{\omega[\vec{v} \mapsto \mathsf{Init}(\vec{v})][\mathsf{thread} \mapsto \alpha],\sigma} &= [\![\vec{e}_2[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega[\vec{v} \mapsto \mathsf{Init}(\vec{v})][\mathsf{thread} \mapsto \alpha],\sigma} \\
&= [\![\vec{e}_2]\!]_{\mathcal{G}}^{\omega[\vec{v} \mapsto \mathsf{Init}(\vec{v})][\mathsf{thread} \mapsto \alpha],\sigma(\alpha),\tau} \\
&= [\![\vec{e}_2]\!]_{\mathcal{G}}^{\omega',\sigma(\alpha),\tau} .
\end{aligned}
$$

Since $\langle T', \sigma' \rangle$ with $T' = \{(\alpha, \tau', stm)\}$ and $\tau' = \tau[\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}]$ is an initial global configuration of $prog'$ after the observation at the beginning of the main statement, it is reachable, and the initial condition for the global invariant is satisfied. The cases for p_3 and I are similar to that of GI, where we additionally use the lifting substitution Lemma 1 to show that $[\![P_3(z)]\!]_{\mathcal{G}}^{\omega',\sigma'} = [\![p_3]\!]_{\mathcal{L}}^{\omega',\sigma'(\alpha),\tau'}$. □

Lemma 14 (Local correctness: Assignment). *The proof outline $prog'$ satisfies the conditions of local correctness from Definition 13.*

Proof (of Lemma 14). Let c be a class of $prog'$ with class invariant I, $\omega \in \Omega$, $\sigma_{inst} \in \Sigma_{inst}$, and $\tau \in \Sigma_{loc}$ with $\sigma_{inst}(\mathsf{this}) = \alpha$. Assume a multiple assignment $\{p_1\}\,\vec{y} := \vec{e}\{p_2\}$ in c which is not the observation of communication or object creation. We have to show that

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p_1 \rightarrow p_2[\vec{e}/\vec{y}]\ .$$

From $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p_1$ it follows by the definition of the annotation that there is a reachable $\langle \grave{T}, \grave{\sigma} \rangle$ with $\grave{\sigma}(\alpha) = \sigma_{inst}$ and $(\alpha, \tau, \vec{y} := \vec{e}; stm) \in \grave{T}$. Executing the local configuration in $\langle \grave{T}, \grave{\sigma} \rangle$ leads to a reachable global configuration $\langle \acute{T}, \acute{\sigma} \rangle$ with $\acute{\sigma}(\alpha) = \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}]$ and $(\alpha, \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}], stm) \in \acute{T}$. Thus by the definition of the annotation for $prog'$ we have

$$\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}], \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}] \models_{\mathcal{L}} p_2\ ,$$

and further with the substitution Lemma 3 $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p_2[\vec{e}/\vec{y}]$, as required. $\square$

Lemma 15 (Local correctness: Exception handling). *The proof outline $prog'$ satisfies the conditions of local correctness from Definition 14.*

Proof (of Lemma 15). Let stm be a statement of the form $\mathsf{try}\ \langle \vec{y}_{\mathsf{try}} := \vec{e}_{\mathsf{try}} \rangle^{try}\, stm_0;$ $\mathsf{catch}\,(c_1\, u_1)\, stm_1; \ldots; \mathsf{catch}\,(c_n\, u_n)\, stm_n;\ \mathsf{finally}\ \langle \vec{y}_{\mathsf{fin}} := \vec{e}_{\mathsf{fin}} \rangle^{fin}\ stm_{n+1}\ \mathsf{yrt}\ \langle \vec{y}_{\mathsf{yrt}} := \vec{e}_{\mathsf{yrt}} \rangle^{yrt}$ in a class c. We show that for all $\grave{\omega}$, $\grave{\sigma}_{inst}$, and $\grave{\tau}$,

$$\begin{aligned}\grave{\omega}, \grave{\sigma}_{inst}, \grave{\tau} \models_{\mathcal{L}} pre(stm) \rightarrow\ & pre(\vec{y}_{\mathsf{try}} := \vec{e}_{\mathsf{try}})[\mathsf{exc} \circ \mathsf{null}/\mathsf{exc}] \wedge \\ & pre(stm_0)[\vec{e}_{\mathsf{try}}/\vec{y}_{\mathsf{try}}][\mathsf{exc} \circ \mathsf{null}/\mathsf{exc}]\ .\end{aligned}$$

From $\grave{\omega}, \grave{\sigma}_{inst}, \grave{\tau} \models_{\mathcal{L}} pre(stm)$ it follows by the definition of the annotation that there is a reachable $\langle \grave{T}, \grave{\sigma} \rangle$ with $\grave{\sigma}(\alpha) = \grave{\sigma}_{inst}$ and $(\alpha, \grave{\tau}, stm; stm') \in \grave{T}$. Executing the exception throwing in the above local configuration in $\langle \grave{T}, \grave{\sigma} \rangle$ updates the local state to $\check{\tau} = \grave{\tau}[\mathsf{exc} \mapsto [\![\mathsf{exc}]\!]_{\mathcal{E}}^{\grave{\sigma}_{inst}, \grave{\tau}} \circ null]$. The corresponding observation completes the computation step and leads to a reachable global configuration $\langle \acute{T}, \acute{\sigma} \rangle$ with $\acute{\sigma} = \grave{\sigma}[\alpha.\grave{\sigma}_{inst}[\vec{y}_{\mathsf{try}} \mapsto [\![\vec{e}_{\mathsf{try}}]\!]_{\mathcal{E}}^{\grave{\sigma}_{inst}, \check{\tau}}] \mapsto]$, $\acute{\tau} = \grave{\tau}[\vec{y}_{\mathsf{try}} \mapsto [\![\vec{e}_{\mathsf{try}}]\!]_{\mathcal{E}}^{\grave{\sigma}_{inst}, \check{\tau}}]$, and $(\alpha, \acute{\tau}, stm_0; \mathsf{catch}\,(c_1\, u_1)\, stm_1; \ldots; \mathsf{catch}\,(c_n\, u_n)\, stm_n; \mathsf{finally}\ \langle \vec{y}_{\mathsf{fin}} := \vec{e}_{\mathsf{fin}} \rangle^{fin}\, stm_{n+1}$ $\mathsf{yrt}) \in \acute{T}$.

Thus by the definition of the annotation for $prog'$ we have

$$\grave{\omega}, \acute{\sigma}_{inst}, \acute{\tau} \models_{\mathcal{L}} pre(stm_0)\ ,$$

and further with the substitution Lemma 3

$$\grave{\omega}, \grave{\sigma}_{inst}, \grave{\tau} \models_{\mathcal{L}} pre(stm_0)[\vec{e}_{\mathsf{fin}}/\vec{y}_{\mathsf{fin}}][\mathsf{exc} \circ \mathsf{null}/\mathsf{exc}]\ .$$

Note that the annotation may not contain free logical variables.

The case for the precondition of the observation is similar: By definition we have $\grave{\omega}, \grave{\sigma}_{inst}, \check{\tau} \models_{\mathcal{L}} pre(\vec{y}_{\mathsf{try}} := \vec{e}_{\mathsf{try}})$, and thus $\grave{\omega}, \grave{\sigma}_{inst}, \grave{\tau} \models_{\mathcal{L}} pre(\vec{y}_{\mathsf{try}} := \vec{e}_{\mathsf{try}})[\mathsf{exc} \circ \mathsf{null}/\mathsf{exc}]$, as required.

The other cases are similar. The antecedents of the conditions assure reachability and enabledness; we use the local substitution lemma to show the required properties. □

Lemma 16 (Interference freedom). *The proof outline prog′ satisfies the conditions for interference freedom from Definition 15.*

Proof (of Lemma 16). Assume an arbitrary assignment $\vec{y} := \vec{e}$ with precondition p in class c with class invariant I, and an arbitrary assertion q at a control point in the same class. We show the verification condition from Equation (32) on page 52

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p \wedge q' \wedge \mathsf{interleavable}(q, \vec{y} := \vec{e}) \rightarrow q'[\vec{e}/\vec{y}] \ ,$$

for some logical environment ω together with some instance and local states σ_{inst} and τ, where q' denotes q with all local variables u replaced by some fresh local variables u'.

Let $\alpha = \sigma_{inst}(\mathsf{this})$, and assume first that $\vec{y} := \vec{e}$ is not the observation of communication, object creation, or exception throwing or handling. The first clause $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ implies that there exists a computation reaching $\langle \grave{T}_p, \grave{\sigma}_p \rangle$ with $\grave{\sigma}_p(\alpha) = \sigma_{inst}$, and a configuration $(\alpha, \tau, \vec{y} := \vec{e}; stm'_p) \in \grave{T}_p$.

From $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'$ we get by renaming back the local variables that $\omega, \sigma_{inst}, \tau' \models_{\mathcal{L}} q$ for $\tau'(u) = \tau(u')$ for all local variables u in q. Let q be the precondition of the statement stm_q. Note that q is an assertion at a control point. Applying the annotation definition we conclude that there is a reachable $\langle \grave{T}_q, \grave{\sigma}_q \rangle$ with $\grave{\sigma}_q(\alpha) = \sigma_{inst} = \grave{\sigma}_p(\alpha)$ and $(\alpha, \tau', stm_q; stm'_q) \in \grave{T}_q$. The local merging Lemma 9 implies that $(\alpha, \tau', stm_q; stm'_q) \in \grave{T}_p$.

Let $\langle \acute{T}_p, \acute{\sigma}_p \rangle$ result from $\langle \grave{T}_p, \grave{\sigma}_p \rangle$ by executing the enabled local configuration $(\alpha, \tau, \vec{y} := \vec{e}; stm'_p)$. We have $\acute{\sigma}_p(\alpha) = \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}]$. From the assumption $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{interleavable}(q, \vec{y} := \vec{e})$ we get that $(\alpha, \tau', stm_q; stm'_q)$ is not the executing configuration, and thus $(\alpha, \tau', stm_q; stm'_q) \in \acute{T}_p$.

According to the annotation definition $\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}], \tau' \models_{\mathcal{L}} q$, and after renaming the local variables of q also $\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}], \tau \models_{\mathcal{L}} q'$. Due to renaming, no local variables of q' occur in $\vec{y}$, implying

$$\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}], \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}] \models_{\mathcal{L}} q' \ .$$

Finally, by the substitution Lemma 3 we get $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'[\vec{e}/\vec{y}]$.

If the assignment observes object creation, communication, or exception throwing or handling, the proof is similar. For object creation, $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ implies that there exists a computation reaching $\langle \grave{T}_p, \grave{\sigma}_p \rangle$ with $\grave{\sigma}_p(\alpha) = \sigma_{inst}$, and an enabled configuration $(\alpha, \tau_p, stm_p; stm'_p) \in \grave{T}_p$, where stm_p is of the form

$u := \mathsf{new}; \langle \vec{y} := \vec{e} \rangle^{new}$. The local state τ_p is $\tau[u \mapsto v]$ for some value v, such that the local configuration is enabled to create $\tau(u)$. Directly after creation, the creator local configuration has the local state τ and executes its observation resulting in the local state $\tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}]$ and instance state $\sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}]$. Note that σ_{inst} is not influenced by the object creation itself. Again, the $\mathsf{interleavable}$ predicate assures that $(\alpha, \tau', stm_q; stm'_q)$ is not the executing configuration, and we get $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'[\vec{e}/\vec{y}]$ as above.

The other cases for observations of communication, object creation, or exception throwing and handling are analogous. In the case of caller observation in a self-communication, the restrictions on the augmentation imply that $\vec{y} := \vec{e}$ does not change the values of instance variables, and the requirement follows directly from the assumptions. If p is the precondition of a callee observation at the beginning of a method body, then the annotation assure that the invocation of the method is enabled in $\langle \grave{T}_p, \grave{\sigma}_p \rangle$ such that τ is the local state of the callee directly after communication but before observation. Note that for self-communication, the caller part does not change the instance state. Thus the only update of the instance state of α is given by the effect of $\vec{y} := \vec{e}$. Again, the $\mathsf{interleavable}$ predicate assures that $(\alpha, \tau', stm_q; stm'_q)$ is neither the caller nor the callee, and thus $(\alpha, \tau', stm_q; stm'_q) \in \acute{T}_p$. We get $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'[\vec{e}/\vec{y}]$ as above.

Validity of the verification condition 31 for the class invariant is similar, where we additionally use the fact that the class invariant refers to instance variables only. $\square$

Lemma 17 (Cooperation test: Communication). *The proof outline prog′ satisfies the verification conditions of the cooperation test for communication of Definition 16.*

Proof (of Lemma 17). We distinguish on the kind of communication starting with the verification condition for synchronized method invocation.

Case: CALL

Let $\{p_1\}\, u_{ret} := e_0.m(\vec{e}); \{p_2\}^{!call} \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \{p_3\}^{wait}$ be a statement in a class c of $prog'$ with e_0 of type c', where method $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ of c' is synchronized with body $\{q_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{q_3\}\, stm$, formal parameters $\vec{u}$, local variables without the formal parameters given by $\vec{v}$, and let $q_1 = I_{c'}$ be the callee class invariant. Assume

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} GI \wedge P_1(z) \wedge Q'_1(z') \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}$$

for distinct and fresh $z \in LVar^c$ and $z' \in LVar^{c'}$, and where comm is $E_0(z) = z' \wedge (z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread})$. Note that for completeness we don't need the information stored in the caller class invariant. By definition of the global invariant, the assumption $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} GI$ implies that there exists a reachable $\langle T, \sigma \rangle$ with

$$dom(\grave{\sigma}) = dom(\sigma) \text{ and } \grave{\sigma}(\gamma)(\mathsf{h}_{comm}) = \sigma(\gamma)(\mathsf{h}_{comm}) \text{ for all } \gamma \in dom(\sigma)\,.$$

Assuming $\grave{\omega}(z)=\alpha$ as caller identity, $\grave{\omega},\grave{\sigma}\models_{\mathcal{G}} P_1(z)$ implies $\grave{\omega},\grave{\sigma}(\alpha),\grave{\tau}_1\models_{\mathcal{L}} p_1$ by the substitution Lemma 1, for some local state $\grave{\tau}_1$ with $\grave{\tau}_1(u)=\grave{\omega}(u)$ for all local variables u occurring in p_1. By the annotation definition there exists a reachable configuration $\langle T_1,\sigma_1\rangle$ such that

$$\sigma_1(\alpha)=\grave{\sigma}(\alpha) \text{ and } (\alpha,\grave{\tau}_1,u_{ret}:=e_0.m(\vec{e});\langle\vec{y}_1:=\vec{e}_1\rangle^{!call}\,stm_1)\in T_1\,.$$

Recall that $\sigma(\gamma)(\mathsf{h}_{comm})=\grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma\in dom(\sigma)$, and especially for the caller $\sigma(\alpha)(\mathsf{h}_{comm})=\grave{\sigma}(\alpha)(\mathsf{h}_{comm})=\sigma_1(\alpha)(\mathsf{h}_{comm})$. Using the global merging Lemma 10 applied to $\langle T_1,\sigma_1\rangle$ and $\langle T,\sigma\rangle$ we get that there is a reachable $\langle T',\sigma'\rangle$ with $dom(\sigma')=dom(\sigma)$ and

$$\sigma'(\alpha)=\sigma_1(\alpha) \text{ and } \sigma'(\gamma)=\sigma(\gamma) \text{ for all } \gamma\in dom(\sigma)\backslash\{\alpha\}\,.$$

Furthermore, $(\alpha,\grave{\tau}_1,u_{ret}:=e_0.m(\vec{e});\langle\vec{y}_1:=\vec{e}_1\rangle^{!call}\,stm_1)\in T_1$, $\sigma_1(\alpha)=\sigma'(\alpha)$, and the local merging Lemma 9 implies that

$$(\alpha,\grave{\tau}_1,u_{ret}:=e_0.m(\vec{e});\langle\vec{y}_1:=\vec{e}_1\rangle^{!call}\,stm_1)\in T'\,.$$

Let $\beta=\grave{\omega}(z')$ be the callee object. In case of a self-call, i.e., for $\alpha=\beta$, we directly get that $\langle T'',\sigma''\rangle=\langle T',\sigma'\rangle$ is a reachable configuration such that $\sigma''(\alpha)=\grave{\sigma}(\alpha)$, $\sigma''(\gamma)(\mathsf{h}_{comm})=\grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma\in dom(\grave{\sigma})$, and $(\alpha,\grave{\tau}_1,u_{ret}:=e_0.m(\vec{e});\langle\vec{y}_1:=\vec{e}_1\rangle^{!call}\,stm_1)\in T''$.

Otherwise, the assumption $\grave{\omega},\grave{\sigma}\models_{\mathcal{G}} I_{c'}(z')$ implies $\grave{\omega},\grave{\sigma}(\beta),\tau_2\models_{\mathcal{L}} I_{c'}$ for some local state τ_2. Note that the class invariant contains instance variables, only. By definition of the class invariant, there is a reachable global configuration $\langle T_2,\sigma_2\rangle$ such that

$$\sigma_2(\beta)=\grave{\sigma}(\beta)\,.$$

We need to fall back upon the two merging lemmas once more to obtain a common reachable configuration: Analogously to the caller part, the global merging Lemma 10 applied to $\langle T_2,\sigma_2\rangle$ and $\langle T',\sigma'\rangle$ yields that there is a reachable configuration $\langle T'',\sigma''\rangle$ with $dom(\sigma'')=dom(\sigma')$ and

$$\sigma''(\beta)=\sigma_2(\beta) \text{ and } \sigma''(\gamma)=\sigma'(\gamma) \text{ for all } \gamma\in dom(\sigma')\backslash\{\beta\}\,.$$

Now, $(\alpha,\grave{\tau}_1,u_{ret}:=e_0.m(\vec{e});\langle\vec{y}_1:=\vec{e}_1\rangle^{!call}\,stm_1)\in T'$, $\sigma''(\alpha)=\sigma'(\alpha)$, and the local merging Lemma 9 implies that the local configuration $(\alpha,\grave{\tau}_1,u_{ret}:=e_0.m(\vec{e});\langle\vec{y}_1:=\vec{e}_1\rangle^{!call}\,stm_1)$ is in T''.

Thus $\langle T'',\sigma''\rangle$ is a reachable configuration with $\sigma''(\alpha)=\grave{\sigma}(\alpha)$, $\sigma''(\beta)=\grave{\sigma}(\beta)$, $\sigma''(\gamma)(\mathsf{h}_{comm})=\grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma\in dom(\grave{\sigma})$, and $(\alpha,\grave{\tau}_1,u_{ret}:=e_0.m(\vec{e});\langle\vec{y}_1:=\vec{e}_1\rangle^{!call}\,stm_1)\in T''$.

With the antecedent $\grave{\omega},\grave{\sigma}\models_{\mathcal{G}} z'.\mathsf{lock}=\mathsf{free}\vee\mathsf{thread}(z'.\mathsf{lock})=\mathsf{thread}$ of the cooperation test we get $\grave{\sigma}(\beta)(\mathsf{lock})=\mathit{free}\vee\mathit{thread}(\grave{\sigma}(\beta)(\mathsf{lock}))=\grave{\tau}_1(\mathsf{thread})$. With $\grave{\sigma}(\beta)=\sigma''(\beta)$ and Lemma 7 we get $\neg owns(T''\backslash\{\xi\},\beta)$, where ξ is the stack with $(\alpha,\grave{\tau}_1,u_{ret}:=e_0.m(\vec{e});\langle\vec{y}_1:=\vec{e}_1\rangle^{!call}\,stm_1)$ on top. Furthermore, $\grave{\omega},\grave{\sigma}\models_{\mathcal{G}}\mathsf{comm}$ implies $\grave{\omega},\grave{\sigma}\models_{\mathcal{G}} E_0(z)=z'$, and by the lifting substitution lemma $[\![e_0]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}\sigma'_{inst1}}=$

$[\![e_0]\!]_{\mathcal{E}}^{\sigma''(\alpha),\grave{\tau}_1} = \omega(z') = \beta$. This means, the invocation of method m of β is enabled in the local configuration $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call}\, stm_1)$ in $\langle T'', \sigma'' \rangle$.

The definition of the augmentation, and $\sigma''(\alpha) = \grave{\sigma}(\alpha)$ gives

$$\grave{\omega}, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_2 \,,$$

which by the substitution Lemma 1 and with the definition of $\grave{\tau}_1$ yields $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} P_2(z)$. Due to the renaming mechanism we get

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} P_2(z) \circ f_{comm}$$

for $f_{comm} = [\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}']$. For the precondition of the method body, the annotation definition implies

$$\grave{\omega}, \grave{\sigma}(\beta), \check{\tau}_2 \models_{\mathcal{L}} q_2$$

with $\check{\tau}_2 = \tau_{init}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}_1}]$. For the actual parameters we obtain by the substitution Lemma 1 $[\![\vec{E}(z)]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} = [\![\vec{e}]\!]_{\mathcal{L}}^{\grave{\omega},\grave{\sigma}(\alpha),\grave{\tau}_1} = [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}_1}$, and further with the same lemma

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} Q_2'(z')[\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}']$$

as required by the cooperation test.

Directly after communication we have a global configuration with still the same global state σ''. The caller observation evolves its own local state to $\acute{\tau}_1 = \grave{\tau}_1[\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma''(\alpha),\grave{\tau}_1}]$, and the global state to $\check{\sigma} = \sigma''[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma''(\alpha),\grave{\tau}_1}]$. Finally, the callee observation changes the global state to $\acute{\sigma} = \check{\sigma}[\beta.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\check{\sigma}(\beta),\check{\tau}_2}]$, where its own local state is updated to $\acute{\tau}_2 = \check{\tau}_2[\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\check{\sigma}(\beta),\check{\tau}_2}]$. According to the annotation definition we get

$$\grave{\omega}, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_3, \quad \grave{\omega}, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} q_3, \quad \text{and} \quad \grave{\omega}, \acute{\sigma} \models_{\mathcal{G}} GI \,.$$

Let $\acute{\omega} = \grave{\omega}[\vec{v}' \mapsto Init(\vec{v})][\vec{u}' \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}_1}][\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}_1}][\vec{y}_2' \mapsto [\![\vec{e}_2']\!]_{\mathcal{E}}^{\check{\sigma}(\beta),\check{\tau}_2}]$. The lifting lemma implies $\acute{\omega}, \acute{\sigma} \models_{\mathcal{G}} GI \wedge P_3(z) \wedge Q_3'(z')$; with the global substitution lemma finally

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} (GI \wedge P_3(z) \wedge Q_3'(z'))[\vec{E}_2'(z')/z'.\vec{y}_2'][\vec{E}_1(z)/z.\vec{y}_1][\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}'] \,,$$

and thus the cooperation test is satisfied for the invocation of synchronous methods.

The case for non-synchronized methods is analogous, where the antecedent $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ is dropped.

Case: CALL$_{monitor}$

This case is similar to the above one of CALL, where for the invocation of a method $m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$, the assertion comm is given by $E_0(z) = z' \wedge \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$, implying $owns(\xi, \beta)$ for the caller thread ξ and the callee object β.

Case: $\textsc{Call}_{start}$
Enabledness of starting the thread of an object β requires $\neg started(T'', \beta)$. Due to the definition of comm, we have additionally $\acute{\omega}, \sigma'' \models_{\mathcal{G}} \neg z'.\mathsf{started}$, which implies $\neg\sigma''(\beta)(\mathsf{started})$. We get enabledness by Lemma 8.

Case: $\textsc{Call}^{skip}_{start}$
The enabledness argument is similar for $\textsc{Call}^{skip}_{start}$, where we use $\acute{\omega}, \sigma'' \models_{\mathcal{G}} z'.\mathsf{started}$ to imply the enabledness predicate $started(T'', \beta)$.

Case: $\textsc{Return}$
For return, the construction of $\langle T'', \sigma'' \rangle$ is similar, where we get instead of the enabledness of the caller that the callee configuration $(\beta, \acute{\tau}_2, \mathsf{return}\ e_{ret}; \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret})$ is in $\langle T'', \sigma'' \rangle$, and thus enabled to execute.

Case: $\textsc{Return}_{wait}$
In this case we additionally have to show $\neg owns(T'', \beta)$, which we get from the comm assertion implying $\acute{\omega}, \acute{\sigma} \models_{\mathcal{G}} z'.\mathsf{lock} = \mathsf{free}$ and using Lemma 7.

Case: $\textsc{Return}_{run}$
Since the run-method cannot be invoked directly, we conclude that the executing local configuration is the only one in its stack, i.e., the transition rule $\textsc{Return}_{run}$ of the semantics can be applied in $\langle T'', \sigma'' \rangle$ to terminate the callee $(\beta, \acute{\tau}_2, \mathsf{return}; \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret})$.

□

Lemma 18 (Cooperation test: Instantiation). *The proof outline prog′ satisfies the verification conditions of the cooperation test for object creation of Definition 17.*

Proof (of Lemma 18). Let $\{p_1\}\, u := \mathsf{new}^c; \{p_2\}^{new} \langle \vec{y} := \vec{e} \rangle^{new} \{p_3\}$ be a statement in class c' of $prog'$, and assume

$$\check{\omega}, \check{\sigma} \models_{\mathcal{G}} z \neq \mathsf{null} \wedge z \neq u \wedge \exists z'.\ \mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u.\ P_1(z)) \downarrow z'$$

with $z \in LVar^{c'}$ and $z' \in LVar^{\mathsf{list\,Object}}$ fresh. Note that we don't need the class invariant of the creator for completeness. We show that

$$\check{\omega}, \check{\sigma} \models_{\mathcal{G}} P_2(z) \wedge I_c(u) \wedge (GI \wedge P_3(z))[\vec{E}(z)/z.\vec{y}]\ .$$

Let $\check{\omega}(z) = \alpha$ and $\check{\omega}(u) = \beta$. According to the semantics of assertions we have that

$$\omega, \check{\sigma} \models_{\mathcal{G}} \mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u.\ P_1(z)) \downarrow z'$$

for some logical environment ω that assigns to z' a sequence of objects from $Val^{\mathsf{Object}}_{null}(\check{\sigma}) = \bigcup_c Val^c_{null}(\check{\sigma})$, and agrees on the values of all other variables with $\check{\omega}$. The assertion $\mathsf{Fresh}(z', u)$ is defined by

$$\mathsf{InitState}(u) \wedge u \notin z' \wedge \forall v.\ v \in z' \vee v = u\ ,$$

where $\mathsf{InitState}(u)$ expands to $u \neq \mathsf{null} \wedge \bigwedge_{x \in IVar_c} u.x = \mathsf{Init}(x)$. Thus, $\omega, \breve{\sigma} \models_{\mathcal{G}} \mathsf{Fresh}(z', u)$ implies that $\beta \in Val^c(\breve{\sigma})$ with $\breve{\sigma}(\beta) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \beta]$, and additionally $Val_{null}^{\mathsf{Object}}(\breve{\sigma}) = \omega(z') \dot{\cup} \{\beta\}$. Let $\grave{\sigma}$ be the global state with domain $Val^{\mathsf{Object}}(\grave{\sigma}) = Val^{\mathsf{Object}}(\breve{\sigma}) \setminus \{\beta\}$ and such that $\grave{\sigma}(\gamma) = \breve{\sigma}(\gamma)$ for all objects $\gamma \in Val^{\mathsf{Object}}(\grave{\sigma})$. Then $\breve{\sigma} = \grave{\sigma}[\beta \mapsto \sigma_{inst}^{init}[\mathsf{this} \mapsto \beta]]$, and from

$$\omega, \breve{\sigma} \models_{\mathcal{G}} (GI \wedge \exists u.\ P_1(z)) \downarrow z'$$

we get with Lemma 2

$$\omega, \grave{\sigma} \models_{\mathcal{G}} GI \wedge \exists u.\ P_1(z)\ .$$

By definition of the annotation, $\omega, \grave{\sigma} \models_{\mathcal{G}} GI$ implies that there is a reachable configuration $\langle \grave{T}_1, \grave{\sigma}_1 \rangle$ such that

$$dom(\grave{\sigma}_1) = dom(\grave{\sigma}) \text{ and } \grave{\sigma}_1(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm}) \text{ for all } \gamma \in dom(\grave{\sigma})\ .$$

The precondition of the object creation statement

$$\omega, \grave{\sigma} \models_{\mathcal{G}} \exists u.\ P_1(z)$$

implies

$$\omega[u \mapsto v], \grave{\sigma} \models_{\mathcal{G}} P_1(z)$$

for some $v \in Val_{null}^{\mathsf{Object}}(\grave{\sigma})$. Applying the lifting Lemma 1 we get that

$$\omega, \grave{\sigma}(\alpha), \grave{\tau} \models_{\mathcal{L}} p_1$$

for a local state $\grave{\tau}$ with $\grave{\tau}(u) = v$ and $\grave{\tau}(v) = \omega(v)$ for all other local variables v. By definition of the annotation, there is a reachable global configuration $\langle \grave{T}_2, \grave{\sigma}_2 \rangle$ such that

$$\grave{\sigma}_2(\alpha) = \grave{\sigma}(\alpha) \text{ and } (\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e} \rangle^{new}\, stm) \in \grave{T}_2\ .$$

Recall that $\grave{\sigma}_1(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\grave{\sigma})$; especially we have $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{comm})$. Using the global merging Lemma 10 applied to the reachable global configurations $\langle \grave{T}_2, \grave{\sigma}_2 \rangle$ and $\langle \grave{T}_1, \grave{\sigma}_1 \rangle$ we get that there is a reachable configuration $\langle \grave{T}_3, \grave{\sigma}_3 \rangle$ with

$$dom(\grave{\sigma}_3) = dom(\grave{\sigma}_1),\ \grave{\sigma}_3(\alpha) = \grave{\sigma}_2(\alpha), \text{ and } \grave{\sigma}_3(\gamma) = \grave{\sigma}_1(\gamma) \text{ for all } \gamma \in dom(\grave{\sigma}_1) \setminus \{\alpha\}.$$

Furthermore, $(\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e} \rangle^{new}\, stm) \in \grave{T}_2$, $\grave{\sigma}_2(\alpha) = \grave{\sigma}_3(\alpha)$, and the local merging Lemma 9 implies that $(\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e} \rangle^{new}\, stm) \in \grave{T}_3$.

So we know that $\langle \grave{T}_3, \grave{\sigma}_3 \rangle$ is a reachable configuration containing the local configuration $(\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e} \rangle^{new}\, stm) \in \grave{T}_3$. With $Val^{\mathsf{Object}}(\grave{\sigma}) = Val^{\mathsf{Object}}(\breve{\sigma}) \setminus \{\beta\}$, $dom(\grave{\sigma}_1) = dom(\grave{\sigma})$, and $dom(\grave{\sigma}_3) = dom(\grave{\sigma}_1)$ we get that $\beta \notin dom(\grave{\sigma}_3)$, i.e., the local configuration is enabled to create the fresh object $\beta = \omega(u)$. With $\grave{\sigma}_3(\alpha) = \grave{\sigma}_2(\alpha) = \breve{\sigma}(\alpha)$ we get

$$\omega, \breve{\sigma}(\alpha), \breve{\tau} \models_{\mathcal{L}} p_2\ ,$$

where $\check{\tau} = \grave{\tau}[u \mapsto \beta]$; with the lifting Lemma 1 together with the definition of $\grave{\tau}$ this means $\omega, \check{\sigma} \models_{\mathcal{G}} P_2(z)$, as required in the cooperation test.

Executing the instantiation in the local configuration $(\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e}\rangle^{new}\, stm)$ in $\langle \grave{T}_3, \grave{\sigma}_3 \rangle$, creating a new object $\beta \notin dom(\grave{\sigma}_3)$, results in $\langle \check{T}_3, \check{\sigma}_3 \rangle$ with $\check{\sigma}_3 = \grave{\sigma}_3[\beta \mapsto \sigma_{inst}^{init}[\mathsf{this} \mapsto \beta]]$; executing the creator observation leads to a reachable $\langle \acute{T}_3, \acute{\sigma}_3 \rangle$ with $\acute{\sigma}_3 = \check{\sigma}_3[\alpha.\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\check{\sigma}_3(\alpha), \check{\tau}}]$ and $(\alpha, \acute{\tau}, stm)$ in $\acute{T}_3$ with $\acute{\tau} = \check{\tau}[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\check{\sigma}_3(\alpha), \check{\tau}}]$.

As $\langle \acute{T}_3, \acute{\sigma}_3 \rangle$ is reachable with $\acute{\sigma}_3(\beta) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \beta] = \check{\sigma}(\beta)$ we know

$$\check{\omega}, \check{\sigma}(\beta), \acute{\tau} \models_{\mathcal{L}} I_c \,.$$

As I_c may not contain local variables, applying the lifting Lemma 1 again with $\omega(u) = \beta$ yields the required condition $\check{\omega}, \check{\sigma} \models_{\mathcal{G}} I_c(u)$ for the class invariant. It remains to show that

$$\check{\omega}, \check{\sigma} \models_{\mathcal{G}} (GI \wedge P_3(z))[\vec{E}(z)/z.\vec{y}] \,.$$

Applying the substitution Lemma 4 and the fact that GI does not contain free logical variables yields

$$\llbracket GI[\vec{E}(z)/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\check{\omega}, \check{\sigma}} = \llbracket GI \rrbracket_{\mathcal{G}}^{\check{\omega}, \acute{\sigma}}$$

with $\acute{\sigma} = \check{\sigma}[\alpha.\vec{y} \mapsto \llbracket \vec{E}(z) \rrbracket_{\mathcal{G}}^{\check{\omega}, \check{\sigma}}]$. Thus we have to show the existence of a reachable configuration with a global state defining the same object domain and communication history values as $\acute{\sigma}$. The configuration $\langle \acute{T}_3, \acute{\sigma}_3 \rangle$ satisfies the above requirements, since, first, it is reachable with

$$\begin{aligned} Val^{\mathsf{Object}}(\acute{\sigma}_3) &= Val^{\mathsf{Object}}(\grave{\sigma}_3) \,\dot{\cup}\, \{\beta\} = Val^{\mathsf{Object}}(\grave{\sigma}_1) \,\dot{\cup}\, \{\beta\} \\ &= Val^{\mathsf{Object}}(\grave{\sigma}) \,\dot{\cup}\, \{\beta\} = Val^{\mathsf{Object}}(\check{\sigma}) = Val^{\mathsf{Object}}(\acute{\sigma}) \,. \end{aligned}$$

Furthermore, $\acute{\sigma}_3(\alpha) = \check{\sigma}_3(\alpha)[\vec{y} \mapsto \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\check{\sigma}_3(\alpha), \check{\tau}}]$, and with $\check{\sigma}_3(\alpha) = \grave{\sigma}_3(\alpha) = \grave{\sigma}_2(\alpha) = \check{\sigma}(\alpha)$ and

$$\llbracket \vec{E}(z) \rrbracket_{\mathcal{G}}^{\check{\omega}, \check{\sigma}} = \llbracket \vec{e}[z/\mathsf{this}] \rrbracket_{\mathcal{G}}^{\check{\omega}, \check{\sigma}} = \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\check{\sigma}(\alpha), \check{\tau}} = \llbracket \vec{e} \rrbracket_{\mathcal{E}}^{\check{\sigma}_3(\alpha), \check{\tau}} \,,$$

we get $\acute{\sigma}_3(\alpha) = \acute{\sigma}(\alpha)$. For the new object, $\acute{\sigma}_3(\beta) = \check{\sigma}_3(\beta) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \beta] = \check{\sigma}(\beta) = \acute{\sigma}(\beta)$. Finally, for all other objects γ different from both α and β from the domain of $\acute{\sigma}$ we have $\acute{\sigma}_3(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}_3(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}_1(\gamma)(\mathsf{h}_{comm}) = \acute{\sigma}(\gamma)(\mathsf{h}_{comm})$.

Similarly for the postcondition p_3 of the observation,

$$\begin{aligned} \llbracket P_3(z)[\vec{E}(z)/z.\vec{y}] \rrbracket_{\mathcal{G}}^{\check{\omega}, \check{\sigma}} &= \llbracket P_3(z) \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} \\ &= \llbracket p_3[z/\mathsf{this}] \rrbracket_{\mathcal{G}}^{\acute{\omega}, \acute{\sigma}} = \llbracket p_3 \rrbracket_{\mathcal{L}}^{\acute{\omega}, \acute{\sigma}(\alpha), \acute{\tau}} = \llbracket p_3 \rrbracket_{\mathcal{L}}^{\acute{\omega}, \acute{\sigma}_3(\alpha), \acute{\tau}} \,. \end{aligned}$$

Thus we have to show the existence of a reachable configuration with a global state defining the same instance state for α as $\acute{\sigma}_3$ and containing the local configuration $(\alpha, \acute{\tau}, stm)$. The configuration $\langle \acute{T}_3, \acute{\sigma}_3 \rangle$ satisfies the above requirements. $\square$

Lemma 19 (Cooperation test: Exception handling). *The proof outline $prog'$ satisfies the verification conditions of the cooperation test for exception handling of Definition 18.*

Proof (of Lemma 19). The proof is analogous to the proof for the cooperation test for communication. Let $u_{ret} := e_0.m(\vec{e})\ \langle stm\rangle^{!call}\{p_1\}^{wait}\{p_2\}^{?ret}\langle \vec{y}_4 := \vec{e}_4\rangle^{?ret}\{p_3\}$ be a statement in a class c with $m \neq \mathsf{start}$ and e_0 of type c', and let $\{q_1\}\ \mathsf{throw}\ e\,\{q_2\}^{throw}\langle \vec{y}_3 := \vec{e}_3\rangle^{throw}$ be a statement in $m(\vec{u})$ of c' which is not in the try-block of any try-catch-finally statement. We have to show that

$$\begin{aligned}\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}}\ & GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm} \\ & \rightarrow (P_2(z) \wedge Q_2'(z')) \circ f_{throw} \wedge (GI \wedge P_3(z)) \circ f_{obs2} \circ f_{obs1} \circ f_{throw}\end{aligned}$$

holds for arbitrary $\dot{\omega}$ and $\dot{\sigma}$, with distinct fresh logical variables $z \in LVar^{c}$ and $z' \in LVar^{c'}$, and with comm given by $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge E'(z') \neq \mathsf{null} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}$. Furthermore, f_{throw} is $[E'(z')/\mathsf{top}]$, f_{obs1} is $[\vec{E}_3'(z')/z'.\vec{y}_3']$, and f_{obs2} is $[\vec{E}_4(z)/z.\vec{y}_4]$.

So assume that the antecedent holds. From $\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} GI$ we get that there exists a reachable $\langle T, \sigma\rangle$ with

$$dom(\dot{\sigma}) = dom(\sigma) \text{ and } \dot{\sigma}(\gamma)(\mathsf{h}_{comm}) = \sigma(\gamma)(\mathsf{h}_{comm}) \text{ for all } \gamma \in dom(\sigma)\,.$$

Assuming $\dot{\omega}(z) = \alpha$ as caller identity, $\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} P_1(z)$ implies $\dot{\omega}, \dot{\sigma}(\alpha), \dot{\tau}_1 \models_{\mathcal{L}} p_1$ by the substitution Lemma 1, for some local state $\dot{\tau}_1$ with $\dot{\tau}_1(u) = \dot{\omega}(u)$ for all local variables u occurring in p_1. By the annotation definition there exists a reachable configuration $\langle T_1, \sigma_1\rangle$ such that

$$\sigma_1(\alpha) = \dot{\sigma}(\alpha) \text{ and } (\alpha, \dot{\tau}_1, \mathsf{receive}\, u_{ret}; \langle \vec{y}_4 := \vec{e}_4\rangle^{?ret} stm_1) \in T_1\,.$$

Recall that $\sigma(\gamma)(\mathsf{h}_{comm}) = \dot{\sigma}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma)$, and especially for the caller $\sigma(\alpha)(\mathsf{h}_{comm}) = \dot{\sigma}(\alpha)(\mathsf{h}_{comm}) = \sigma_1(\alpha)(\mathsf{h}_{comm})$. Using the global merging Lemma 10 applied to $\langle T_1, \sigma_1\rangle$ and $\langle T, \sigma\rangle$ we get that there is a reachable $\langle T', \sigma'\rangle$ with $dom(\sigma') = dom(\sigma)$ and

$$\sigma'(\alpha) = \sigma_1(\alpha) \text{ and } \sigma'(\gamma) = \sigma(\gamma) \text{ for all } \gamma \in dom(\sigma) \backslash \{\alpha\}\,.$$

Furthermore, $(\alpha, \dot{\tau}_1, \mathsf{receive}\, u_{ret}; \langle \vec{y}_4 := \vec{e}_4\rangle^{?ret} stm_1) \in T_1$, $\sigma_1(\alpha) = \sigma'(\alpha)$, and the local merging Lemma 9 implies that

$$(\alpha, \dot{\tau}_1, \mathsf{receive}\, u_{ret}; \langle \vec{y}_4 := \vec{e}_4\rangle^{?ret} stm_1) \in T'\,.$$

Let $\beta = \dot{\omega}(z')$ be the callee object. The assumption $\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} Q_1'(z')$ implies $\dot{\omega}, \dot{\sigma}(\beta), \dot{\tau}_2 \models_{\mathcal{L}} q_1$ with $\dot{\tau}_2(v) = \dot{\omega}(v')$ for all local variables v in q_1. By definition of q_1 there is a reachable global configuration $\langle T_2, \sigma_2\rangle$ such that

$$\sigma_2(\beta) = \dot{\sigma}(\beta) \text{ and } (\beta, \dot{\tau}_2, \mathsf{throw}\, e; \langle \vec{y}_3 := \vec{e}_3\rangle^{throw} stm_2) \in T_2\,.$$

In case of a self-call, i.e., for $\alpha = \beta$, we directly get that $\langle T'', \sigma''\rangle = \langle T', \sigma'\rangle$ is a reachable configuration such that $\sigma''(\alpha) = \dot{\sigma}(\alpha) = \dot{\sigma}(\beta)$, $\sigma''(\gamma)(\mathsf{h}_{comm}) =$

$\grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\grave{\sigma})$, and $(\alpha, \grave{\tau}_1, \mathsf{receive}\, u_{ret}; \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} stm_1) \in T''$. With the local merging lemma we get additionally $(\beta, \grave{\tau}_2, \mathsf{throw}\, e; \langle \vec{y}_3 := \vec{e}_3 \rangle^{throw} stm_2) \in T''$.

Otherwise, for a non-self-call, we need to fall back upon the two merging lemmas once more to obtain a common reachable configuration: Analogously to the caller part, the global merging Lemma 10 applied to $\langle T_2, \sigma_2 \rangle$ and $\langle T', \sigma' \rangle$ yields that there is a reachable configuration $\langle T'', \sigma'' \rangle$ with $dom(\sigma'') = dom(\sigma')$ and

$$\sigma''(\beta) = \sigma_2(\beta) \text{ and } \sigma''(\gamma) = \sigma'(\gamma) \text{ for all } \gamma \in dom(\sigma') \setminus \{\beta\}\,.$$

Now, $(\alpha, \grave{\tau}_1, \mathsf{receive}\, u_{ret}; \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} stm_1) \in T'$, $\sigma''(\alpha) = \sigma'(\alpha)$, and the local merging Lemma 9 implies that the local configuration $(\alpha, \grave{\tau}_1, \mathsf{receive}\, u_{ret}; \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} stm_1)$ is in T''. Similarly, $(\beta, \grave{\tau}_2, \mathsf{throw}\, e; \langle \vec{y}_3 := \vec{e}_3 \rangle^{throw} stm_2) \in T_2$, $\sigma''(\beta) = \sigma_2(\beta)$, and the local merging Lemma 9 implies $(\beta, \grave{\tau}_2, \mathsf{throw}\, e; \langle \vec{y}_3 := \vec{e}_3 \rangle^{throw} stm_2) \in T''$.

Thus $\langle T'', \sigma'' \rangle$ is a reachable configuration with $\sigma''(\alpha) = \grave{\sigma}(\alpha)$, $\sigma''(\beta) = \grave{\sigma}(\beta)$, $\sigma''(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\grave{\sigma})$, $(\alpha, \grave{\tau}_1, \mathsf{receive}\, u_{ret}; \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} stm_1) \in T''$ and $(\beta, \grave{\tau}_2, \mathsf{throw}\, e; \langle \vec{y}_3 := \vec{e}_3 \rangle^{throw} stm_2) \in T''$.

With the antecedent $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \mathsf{comm}$ of the cooperation test we get $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge E'(z') \neq \mathsf{null} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}$, and by the lifting substitution lemma $[\![e_0]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1} = [\![e_0]\!]_{\mathcal{E}}^{\sigma''(\alpha), \grave{\tau}_1} = \grave{\omega}(z') = \beta$. Furthermore, using the same lemma gives $[\![\vec{u}]\!]_{\mathcal{E}}^{\grave{\omega}, \sigma''(\beta), \grave{\tau}_2} = [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\omega}, \sigma''(\alpha), \grave{\tau}_1}$ and $[\![e]\!]_{\mathcal{E}}^{\sigma''(\beta), \grave{\tau}_2} \neq null$. I.e., the values of the formal and actual parameters agree, and thus the augmentation definition and Lemma 6 assures that the local configurations are in caller-callee relationship. Additionally, the value of the exception to be thrown is not the empty reference, and thus the exception throwing is enabled.

The definition of the augmentation, and $\sigma''(\alpha) = \grave{\sigma}(\alpha)$ gives

$$\grave{\omega}, \grave{\sigma}(\alpha), \check{\tau}_1 \models_{\mathcal{L}} p_2\,,$$

with $\check{\tau}_1 = \grave{\tau}_1[\mathsf{top} \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma''(\beta), \grave{\tau}_2}]$, which by the substitution Lemma 1 and with the definition of $\grave{\tau}_1$ implies that $\grave{\omega}[\mathsf{top} \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma''(\beta), \grave{\tau}_2}], \grave{\sigma} \models_{\mathcal{G}} P_2(z)$, i.e.,

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} P_2(z) \circ f_{comm}\,.$$

Since the local state of the callee is not modified during exception throwing, the annotation definition implies $\grave{\omega}, \grave{\sigma}(\beta), \grave{\tau}_2 \models_{\mathcal{L}} q_2$, i.e., $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} Q_2'(z')$. Due to the renaming mechanism we get

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} Q_2'(z') \circ f_{comm}\,.$$

Directly after communication we have a global configuration with still the same global state σ''. The callee observation evolves the global state to $\check{\sigma} = \sigma''[\beta.\vec{y}_3 \mapsto [\![\vec{e}_3]\!]_{\mathcal{E}}^{\sigma''(\beta), \grave{\tau}_2}]$. Finally, the caller observation changes the global state to $\acute{\sigma} = \check{\sigma}[\alpha.\vec{y}_4 \mapsto [\![\vec{e}_4]\!]_{\mathcal{E}}^{\check{\sigma}(\alpha), \check{\tau}_1}]$, where its own local state is updated to $\acute{\tau}_1 = \check{\tau}_1[\vec{y}_4 \mapsto [\![\vec{e}_4]\!]_{\mathcal{E}}^{\check{\sigma}(\alpha), \check{\tau}_1}]$. According to the annotation definition we get

$$\grave{\omega}, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_3 \quad \text{and} \quad \grave{\omega}, \acute{\sigma} \models_{\mathcal{G}} GI\,.$$

Let $\acute{\omega} = \grave{\omega}[\mathsf{top} \mapsto [\![e]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\check{\tau}_2}][\vec{y}_3' \mapsto [\![\vec{e}_3]\!]_{\mathcal{E}}^{\grave{\sigma}(\beta),\check{\tau}_2}][\vec{y}_4 \mapsto [\![\vec{e}_4]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\check{\tau}_1}]$. The lifting lemma implies $\acute{\omega}, \acute{\sigma} \models_{\mathcal{G}} GI \wedge P_3(z)$; with the global substitution lemma finally

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} (GI \wedge P_3(z))[\vec{E}_4(z)/z.\vec{y}_4][\vec{E}_3'/z'.\vec{y}_3'][E'(z')/\mathsf{top}]\,,$$

and thus the cooperation test for exception handling is satisfied for this case. The case for rethrowing is analogous. $\square$

Proof (of Theorem 2). Straightforward using the Lemmas 13, 14, 15, 16, 17, and 19, and 18. $\square$

D Deadlock freedom examples

D.1 Reentrant monitors

$$
\begin{array}{l}
GI \stackrel{def}{=} \\
\quad (\forall (z : Synch).z \neq null \rightarrow \\
\qquad (z.lock = (null, 0) \vee \\
\qquad (\exists (t : Main).owns(t, z.lock) \wedge t.started \wedge t.created = z) \vee \\
\qquad (owns(z, z.lock) \wedge z.started))) \wedge \\
\quad (\forall (t : Main).(t \neq null \wedge \neg t.in\_Synch) \rightarrow (t.created = null \vee not\_owns(t, t.created.lock))) \wedge \\
\quad (\forall (t : Main).t \neq null \rightarrow (\forall (z : Synch).(z \neq null \wedge owns(t, z.lock)) \rightarrow t.created = z))
\end{array}
$$

$I_{Main} \stackrel{def}{=} started$

```
class Main{
    ⟨ boolean in_Synch; ⟩
    ⟨ Synch created; ⟩

    nsync Void run(){
        Synch obj;
        {thread = this ∧ ¬in_Synch ∧ created = null ∧ conf = 0}
        obj = new^Synch; {thread = this ∧ conf = 0}^new ⟨created = obj⟩^new
        {obj ≠ null ∧ obj ≠ this ∧ thread = this ∧ ¬in_Synch ∧ created = obj ∧ conf = 0}
        obj.start();
        {obj ≠ null ∧ obj ≠ this ∧ thread = this ∧ ¬in_Synch ∧ created = obj ∧ conf = 0}
        obj.m1();
        {thread = this ∧ conf = 0}^!call ⟨in_Synch = (if obj = this then in_Synch else true fi)⟩^!call
        {thread = this ∧ created = obj ∧ conf = 0}^wait
        {thread = this ∧ conf = 0}^?ret ⟨in_Synch = (if obj = this then in_Synch else false fi)⟩^?ret
        {thread = this ∧ ¬in_Synch ∧ created = obj ∧ conf = 0}
    }
}

class Synch{

    nsync Void wait(){ {false}^?call {false} return_getlock {false}^!ret }

    sync Void m1(){
        {owns(thread, lock) ∧ depth(lock) = 1}
        m2();
        {owns(thread, lock) ∧ depth(lock) = 1}
    }

    sync Void m2(){
        {owns(thread, lock) ∧ depth(lock) = 2}
    }

    nsync Void run(){
        {thread = this ∧ started ∧ not_owns(thread, lock)}
```

`m1();`
$\{not\_owns(thread, lock)\}$
`}`

`}`

D.2 A simple wait-notify example

$GI \stackrel{def}{=}$
$(\forall(z_1, z_2 : Main).(z_1 \neq null \wedge z_2 \neq null) \to z_1 = z_2)\wedge$
$(\forall(z_1, z_2 : Monitor).(z_1 \neq null \wedge z_2 \neq null) \to z_1 = z_2)\wedge$
$(\forall(z : Main).z \neq null \to$
$(z.started \wedge z.x \geq 0 \wedge z.x \leq 3\wedge$
$(z.x = 0 \to z.created = null \wedge (\forall(z_2 : Monitor).z_2 = null))\wedge$
$(z.x = 1 \to (z.created \neq null \wedge z.created \neq z \wedge z.created.lock = (null, 0) \wedge z.created.x = 0\wedge$
$length(z.created.wait) = 0 \wedge length(z.created.notified) = 0\wedge$
$z.created.counter = 0 \wedge \neg z.created.started))\wedge$
$(z.x = 3 \to z.created \neq null \wedge not\_owns(z, z.created.lock) \wedge z.created.x = 8)\wedge$
$(z.x = 2 \to z.created \neq null)))\wedge$
$(\forall(z_1 : Main).z_1 \neq null \to (\forall(z_2 : Monitor).(z_2 \neq null \wedge owns(z_1, z_2.lock)) \to z_2 = z_1.created))\wedge$

$(\forall(z_1, z_2 : Monitor).(z_1 \neq null \wedge z_2 \neq null \wedge owns(z_1, z_2.lock)) \to (z_1.started \wedge z_2 = z_1))$

$I_{Monitor} \stackrel{def}{=}$
$(\forall(e \in wait \cup notified).e = (creator, 1))\wedge$
$(x = 0 \to (lock = (null, 0) \wedge length(wait) = 0 \wedge length(notified) = 0 \wedge \neg started))\wedge$
$(x = 1 \to (lock = (creator, 1) \wedge length(wait) = 0 \wedge length(notified) = 0 \wedge \neg started))\wedge$
$((x = 2 \vee x = 7) \to (lock = (creator, 1) \wedge length(wait) = 0 \wedge length(notified) = 0 \wedge started))\wedge$
$(x = 3 \to (lock = (null, 0) \wedge length(wait) = 1 \wedge length(notified) = 0 \wedge started))\wedge$
$(x = 4 \to (lock = (this, 1) \wedge ((length(wait) = 1 \wedge length(notified) = 0)\vee$
$(length(wait) = 0 \wedge length(notified) = 1)) \wedge started))\wedge$
$(x = 5 \to (lock = (this, 1) \wedge length(wait) = 0 \wedge length(notified) = 1 \wedge started))\wedge$
$(x = 6 \to (lock = (null, 0) \wedge length(wait) = 0 \wedge length(notified) = 1 \wedge started))\wedge$
$(x = 8 \to (lock = (null, 0) \wedge length(wait) = 0 \wedge length(notified) = 0 \wedge started))$

`class Main{`
$\langle$ `int x;` $\rangle$
$\langle$ `Monitor created;` $\rangle$

`nsync Void run(){`
`Monitor obj;`
$\{x = 0 \wedge thread = this \wedge conf = 0 \wedge started\}$
`obj = new`$^{\mathsf{Monitor}}$`;` $\{thread = this \wedge conf = 0\}^{new}$ $\langle created = obj; x = 1\rangle^{new}$
$\{x = 1 \wedge thread = this \wedge conf = 0 \wedge started \wedge created = obj \wedge obj \neq null\}$
`obj.m1()`
$\{x = 1 \wedge thread = this \wedge conf = 0 \wedge created = obj\}^{!call}$
$\langle x = (\mathsf{if}\ obj = this\ \mathsf{then}\ x\ \mathsf{else}\ 2\ \mathsf{fi})\rangle^{!call}$
$\{x = 2 \wedge thread = this \wedge conf = 0 \wedge created = obj\}^{wait}$
$\{x = 2 \wedge thread = this \wedge conf = 0 \wedge created = obj\}^{?ret}$
$\langle x = (\mathsf{if}\ obj = this\ \mathsf{then}\ x\ \mathsf{else}\ 3\ \mathsf{fi})\rangle^{?ret}$
$\{x = 3 \wedge thread = this \wedge conf = 0 \wedge created = obj\}$
`}`
`}`

`class Monitor{`
$\langle$ `Main creator;` $\rangle$
$\langle$ `int x;` $\rangle$

`nsync Void wait(){`
$\{x = 2 \wedge thread = creator\}^{?call}$ $\langle x = 3\rangle^{?call}$
$\{3 \leq x \wedge x \leq 6 \wedge thread = creator\}$
`return`$_{getlock}$
$\{x = 6 \wedge thread = creator\}^{!ret}$ $\langle x = 7\rangle^{!ret}$
`}`

`nsync Void notify(){`
$\{x = 4 \wedge thread = this \wedge length(wait) = 1\}$
$\langle\rangle$
$\{x = 4 \wedge thread = this \wedge length(wait) = 0\}$
`return`
$\{x = 4 \wedge thread = this \wedge length(wait) = 0\}^{!ret} \quad \langle x = 5\rangle^{!ret}$
`}`

`nsync Void notifyAll(){`
$\{false\} \quad \langle\rangle \quad \{false\}$
`}`

`sync Void m1(){`
$\{x = 0\}^{?call} \quad \langle creator = thread; x = 1\rangle^{?call}$
$\{x = 1 \wedge thread = creator \wedge conf = 0\}$
`start();`
$\{x = 2 \wedge thread = creator\}$
`wait();`
$\{x = 7 \wedge thread = creator\}$
`return`
$\{x = 7 \wedge thread = creator\}^{!ret} \quad \langle x = 8\rangle^{!ret}$
`}`

`nsync Void run(){`
$\{x = 1 \wedge thread = this \wedge \mathsf{caller} = (this, 0, creator)\}^{?call} \quad \langle x = 2\rangle^{?call}$
$\{(x = 2 \vee x = 3) \wedge thread = this \wedge started\}$
`m2()`
$\{(x = 6 \vee x = 7 \vee x = 8) \wedge thread = this\}$
`}`

`sync Void m2(){`
$\{x = 3 \wedge thread = this\}^{?call} \quad \langle x = 4\rangle^{?call}$
$\{x = 4 \wedge thread = this \wedge length(wait) = 1 \wedge started\}$
`notify();`
$\{x = 5 \wedge thread = this\}$
`return`
$\{x = 5 \wedge thread = this\}^{!ret} \quad \langle x = 6\rangle^{!ret}$
`}`
`}`

D.3 A producer-consumer example

$$
\begin{aligned}
GI \stackrel{def}{=}\; & (\forall(p : Producer).(p \neq null \wedge \neg p.outside \wedge p.consumer \neq null) \rightarrow \\
& \quad (p.consumer.lock = (null, 0) \wedge length(p.consumer.wait) = 0 \wedge \\
& \quad p.consumer.producer = null \wedge \neg p.consumer.started \wedge p.consumer.counter = 0)) \wedge \\
& (\forall(p : Producer).(p \neq null \wedge p.consumer \neq null \wedge p.consumer.producer \neq null) \rightarrow p.outside) \wedge \\
& (\forall(c : Consumer).(c \neq null \wedge c.started) \rightarrow (c.producer \neq null \wedge c.producer.started)) \wedge \\
& (\forall(c1, c2 : Consumer).(c1 \neq null \wedge c2 \neq null) \rightarrow c1 = c2)) \wedge \\
& (\exists(p : Producer).p \neq null \wedge (\forall(p2 : Producer).p2 \neq null \rightarrow p2 = p) \wedge \\
& \quad (p.consumer = null \rightarrow (\forall(c : Consumer).c = null))) \wedge \\
& (\forall(c : Consumer).(c \neq null \wedge c.producer \neq null) \rightarrow c.producer.started)
\end{aligned}
$$

$$
\begin{aligned}
I_{\mathsf{Consumer}} \stackrel{def}{=}\; & (lock = (null, 0) \vee (owns(this, lock) \wedge started) \vee owns(producer, lock)) \wedge length(wait) \leq 1
\end{aligned}
$$

`class Producer{`
$\langle$ `Consumer consumer;` $\rangle$
$\langle$ `boolean outside;` $\rangle$

`nsync Void wait(){` $\{false\}^{?call} \quad \{false\}$ $\mathbf{return}_{getlock}$ $\{false\}^{!ret}$ `}`

`nsync Void run(){`
`Consumer c;`

```
            {¬outside ∧ thread = this ∧ consumer = null ∧ started}
            c = new^Consumer;  {thread = this}^new  ⟨consumer = c⟩^new
            {c = consumer ∧ ¬outside ∧ consumer ≠ null ∧ consumer ≠ this∧
              thread = this ∧ started}
            c.produce()  {thread = this}^!call  ⟨outside = (if c = this then outside else true)⟩^!call
            {false}
        }
}

class Consumer{
    int buffer;
    ⟨ Producer producer; ⟩

    nsync Void wait(){
            {owns(thread, lock) ∧ started ∧ length(wait) = 0}^?call
            {started ∧ not_owns(thread, lock) ∧ (thread = this ∨ thread = producer)∧
              (thread ∈ wait ∨ thread ∈ notified)}
            return_getlock
            {started ∧ lock = (null, 0) ∧ thread ≠ null ∧ (thread = this ∨ thread = producer)∧
              thread ∈ notified}^!ret
    }

    nsync Void notify(){
            {owns(thread, lock) ∧ started}
            ⟨⟩
            {owns(thread, lock) ∧ length(wait) = 0}
    }

    nsync Void notifyAll(){ {false} ⟨⟩ }

    sync Void produce(){
            int i;
            {thread ≠ null ∧ producer = null ∧ thread = proj(caller, 1)∧
              length(wait) = 0 ∧ ¬started}^?call
            ⟨producer = proj(caller, 1)⟩^?call
            {owns(thread, lock) ∧ thread = producer ∧ ¬started ∧ conf = 0 ∧ producer ≠ this}
            i=0;
            {owns(thread, lock) ∧ thread = producer ∧ ¬started ∧ conf = 0 ∧ producer ≠ this}
            start();
            {owns(thread, lock) ∧ started ∧ thread = producer}
            while (true){
                {owns(thread, lock) ∧ started ∧ thread = producer}
                //produce i here
                buffer = i;
                {owns(thread, lock) ∧ started ∧ thread = producer}
                notify();
                {started ∧ thread = producer}^wait
                {owns(thread, lock) ∧ started ∧ thread = producer ∧ length(wait) = 0}
                wait()
                {started ∧ thread = producer}^wait
                {owns(thread, lock) ∧ started ∧ thread = producer}
            }
            {false}
            return
            {false}^!ret
    }

    nsync Void run(){
            {¬started ∧ caller = (this, 0, producer)}^?call
            {not_owns(thread, lock) ∧ thread = this ∧ thread ≠ null ∧ started}
            consume()
            {false}
    }

    sync Void consume(){
            int i;

            {thread = this ∧ free_for(thread, lock) ∧ started}^?call
            {owns(thread, lock) ∧ started ∧ thread = this}
            while (true){
                {owns(thread, lock) ∧ started ∧ thread = this}
```

```
            i = buffer;
            //consume i here
            {owns(thread, lock) ∧ started ∧ thread = this}
            notify();
            {started ∧ thread = this}^wait
            {owns(thread, lock) ∧ started ∧ thread = this ∧ length(wait) = 0}
            wait()
            {thread = this}^wait
            {owns(thread, lock) ∧ started ∧ thread = this}
        }
        {false}
        return
        {false}^!ret
    }
}
```